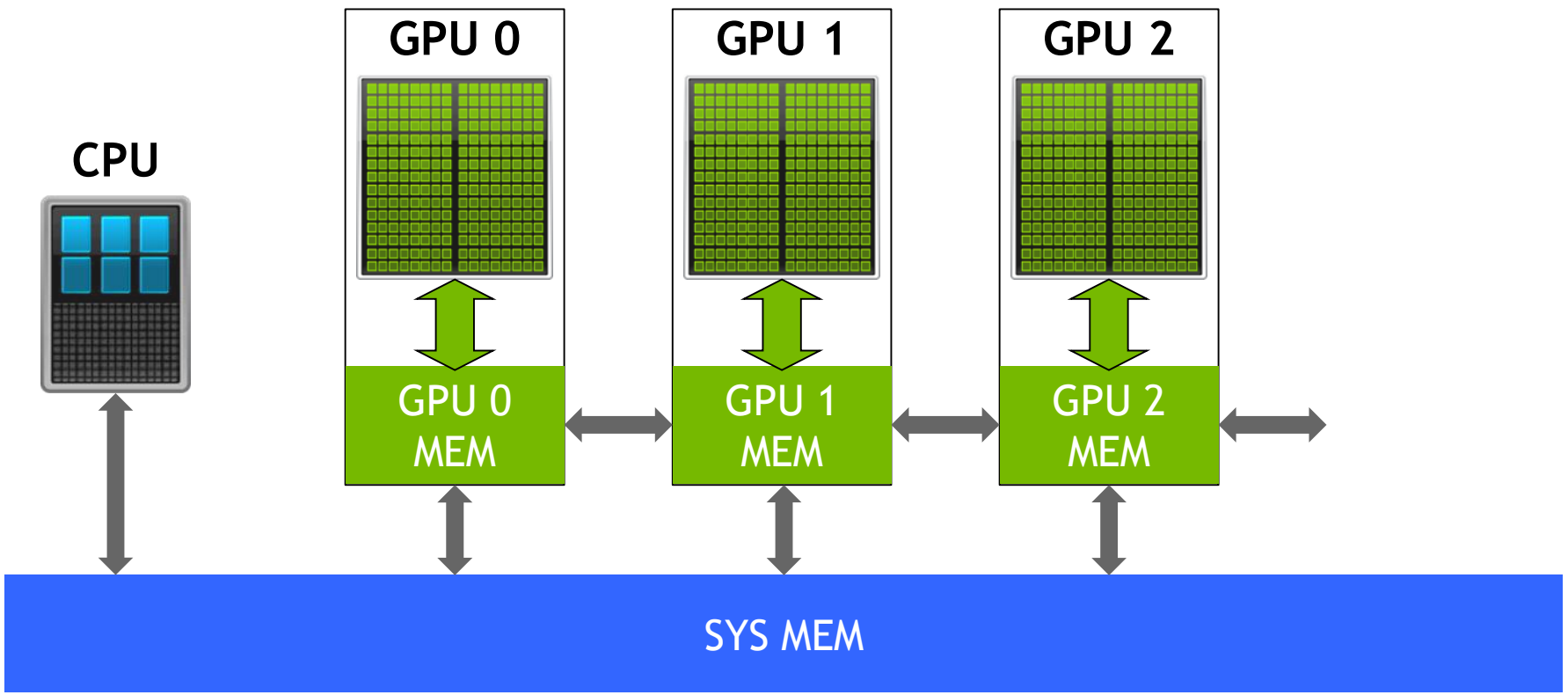
UNIFIED MEMORY ON PASCAL AND VOLTA

Nikolay Sakharnykh - May 10, 2017

NVIDIA

HETEROGENEOUS ARCHITECTURES



UNIFIED MEMORY FUNDAMENTALS

Single Pointer

CPU code

```
void *data;
data = malloc(N);

cpu_func1(data, N);

cpu_func2(data, N);


cpu_func3(data, N);

free(data);
```

GPU code

```
void *data;
data = malloc(N);

cpu_func1(data, N);

gpu_func2<<<...>>>(data, N);
cudaDeviceSynchronize();

cpu_func3(data, N);

free(data);
```

UNIFIED MEMORY FUNDAMENTALS

Single Pointer

Explicit Memory Management

```
void *h_data, *d_data;
h_data = malloc(N);
cudaMalloc(&d_data, N);
cpu_func1(h_data, N);
cudaMemcpy(d_data, h_data, N, ...)
gpu_func2<<<...>>>(data, N);

cudaMemcpy(h_data, d_data, N, ...)
cpu_func3(h_data, N);

free(h_data);
cudaFree(d_data);
```

Unified Memory

```
void *data;
data = malloc(N);

cpu_func1(data, N);

gpu_func2<<<...>>>(data, N);
cudaDeviceSynchronize();

cpu_func3(data, N);

free(data);
```

UNIFIED MEMORY FUNDAMENTALS

Deep Copy Nightmare

Explicit Memory Management

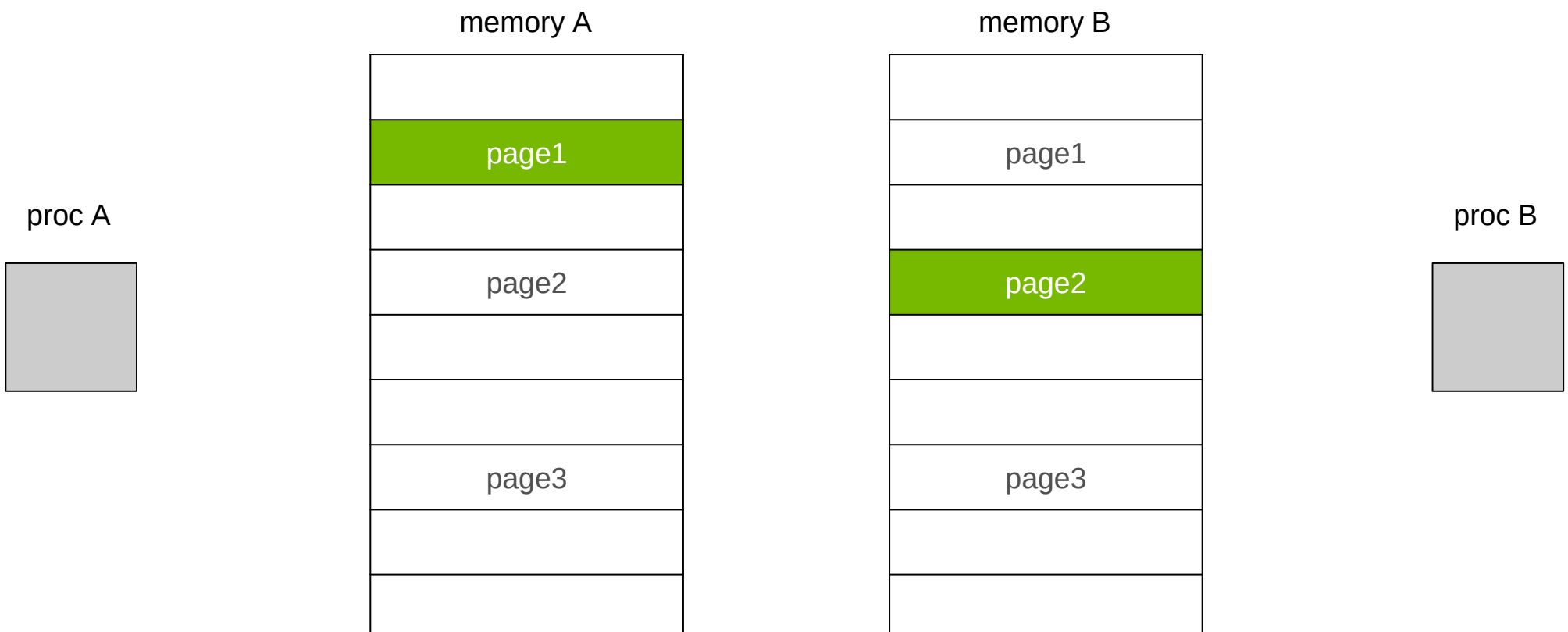
```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
  data[i] = (char*)malloc(N);

char **d_data;
char **h_data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++) {
  cudaMalloc(&h_data2[i], N);
  cudaMemcpy(h_data2[i], h_data[i], N, ...);
}
cudaMalloc(&d_data, N*sizeof(char*));
cudaMemcpy(d_data, h_data2, N*sizeof(char*), ...);

gpu_func<<<...>>>(data, N);
```

Unified Memory

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
  data[i] = (char*)malloc(N);

gpu_func<<<...>>>(data, N);
```

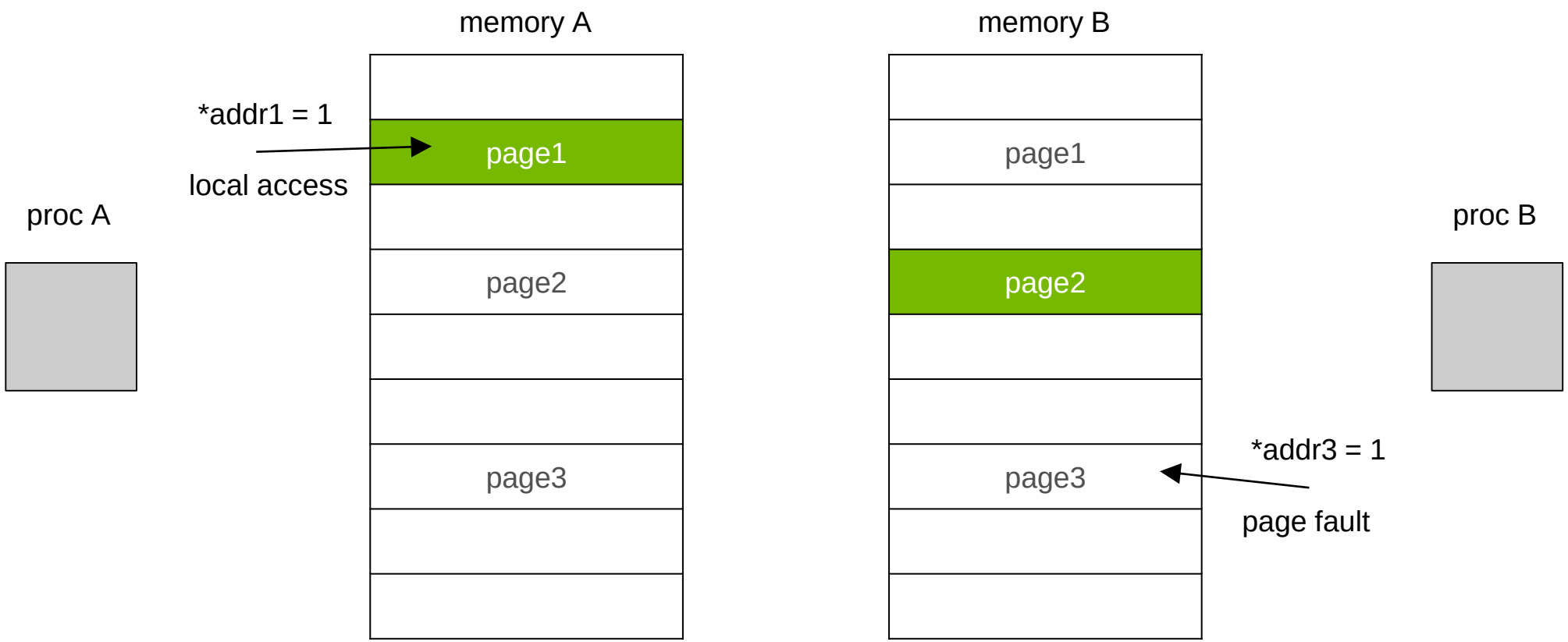
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



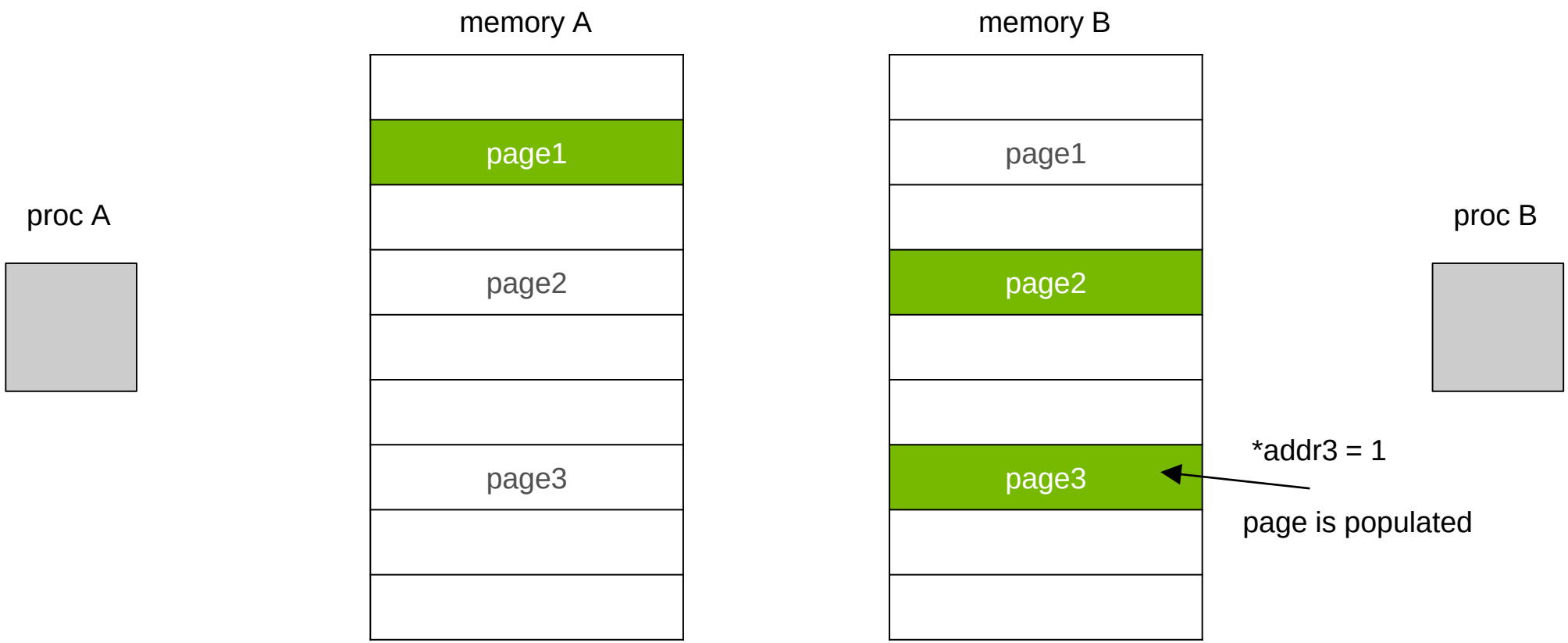
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



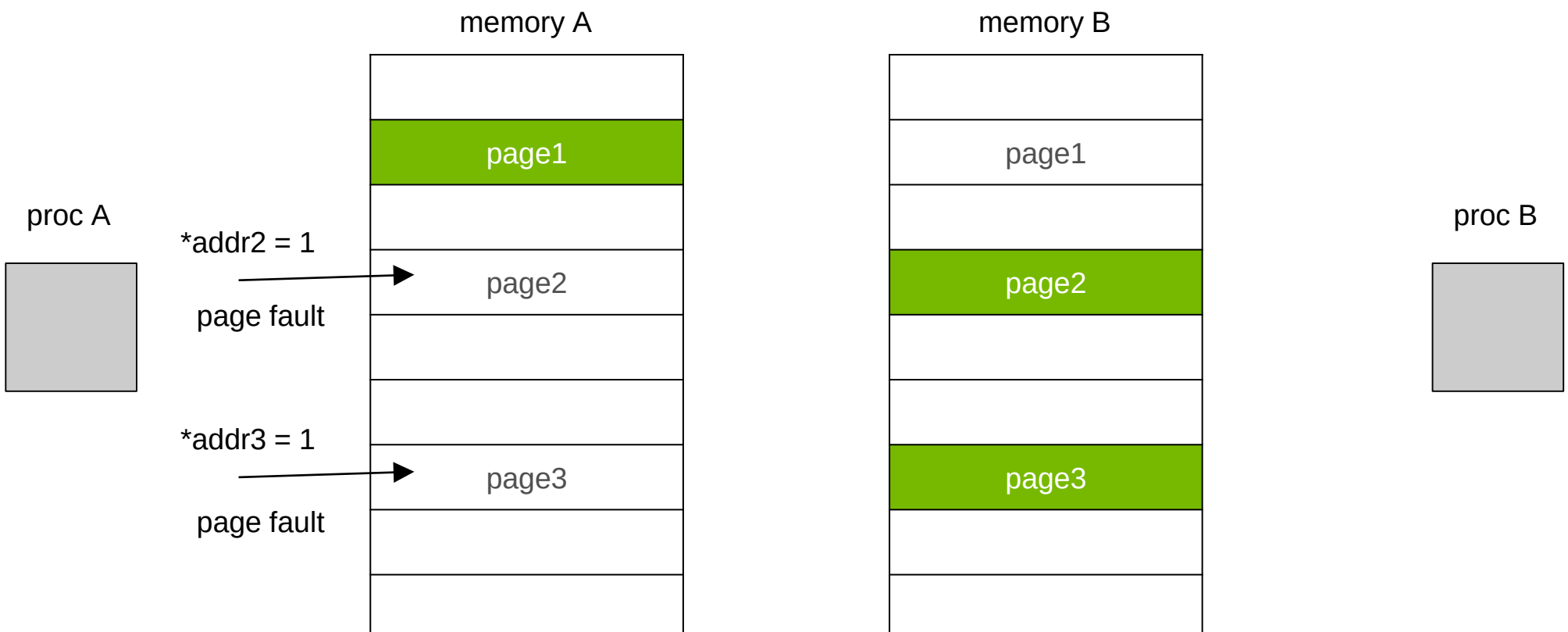
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



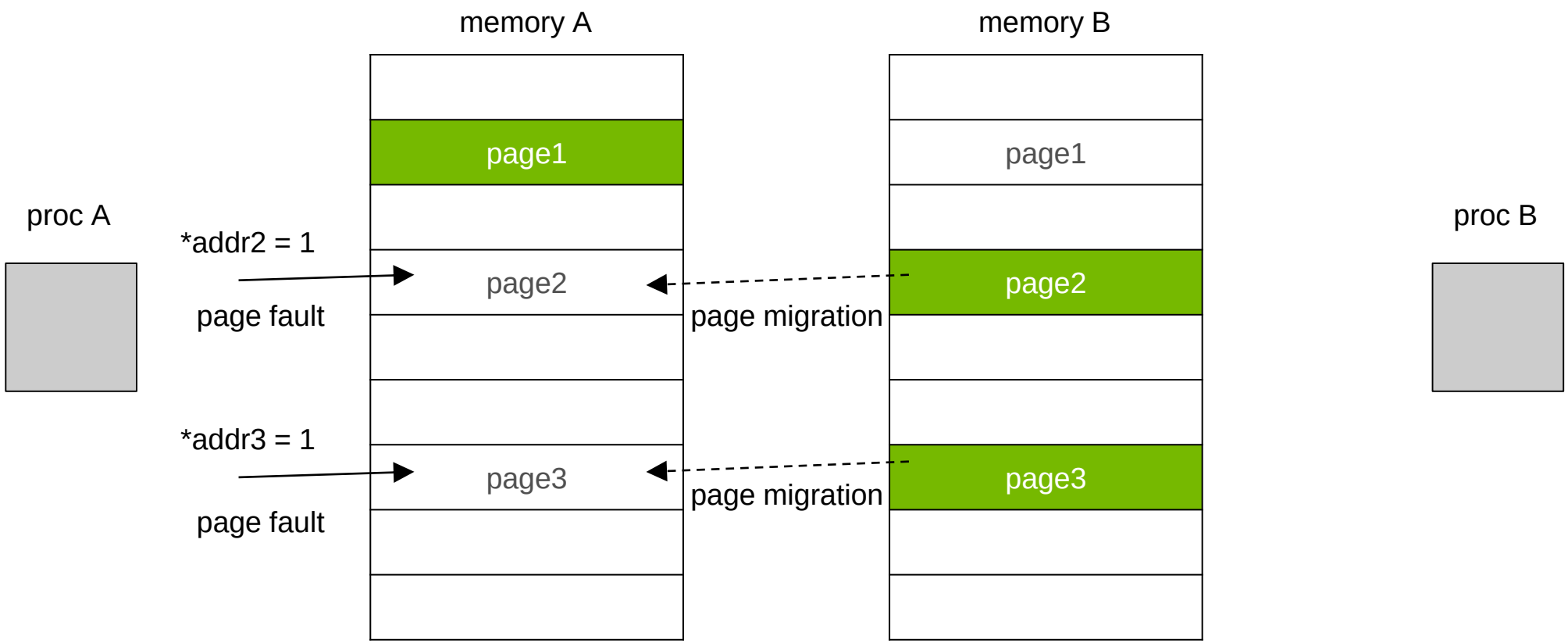
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



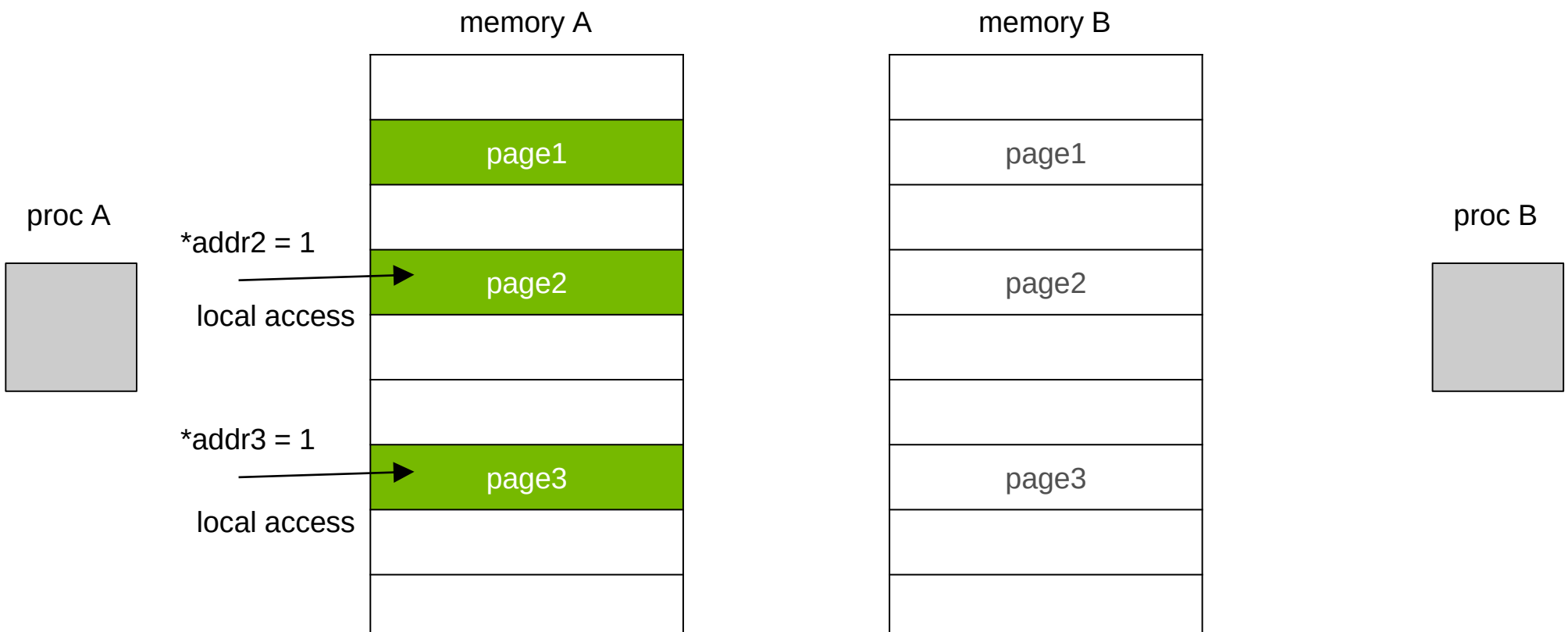
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



UNIFIED MEMORY FUNDAMENTALS

When Is This Helpful?

When it doesn't matter *how* data moves to a processor

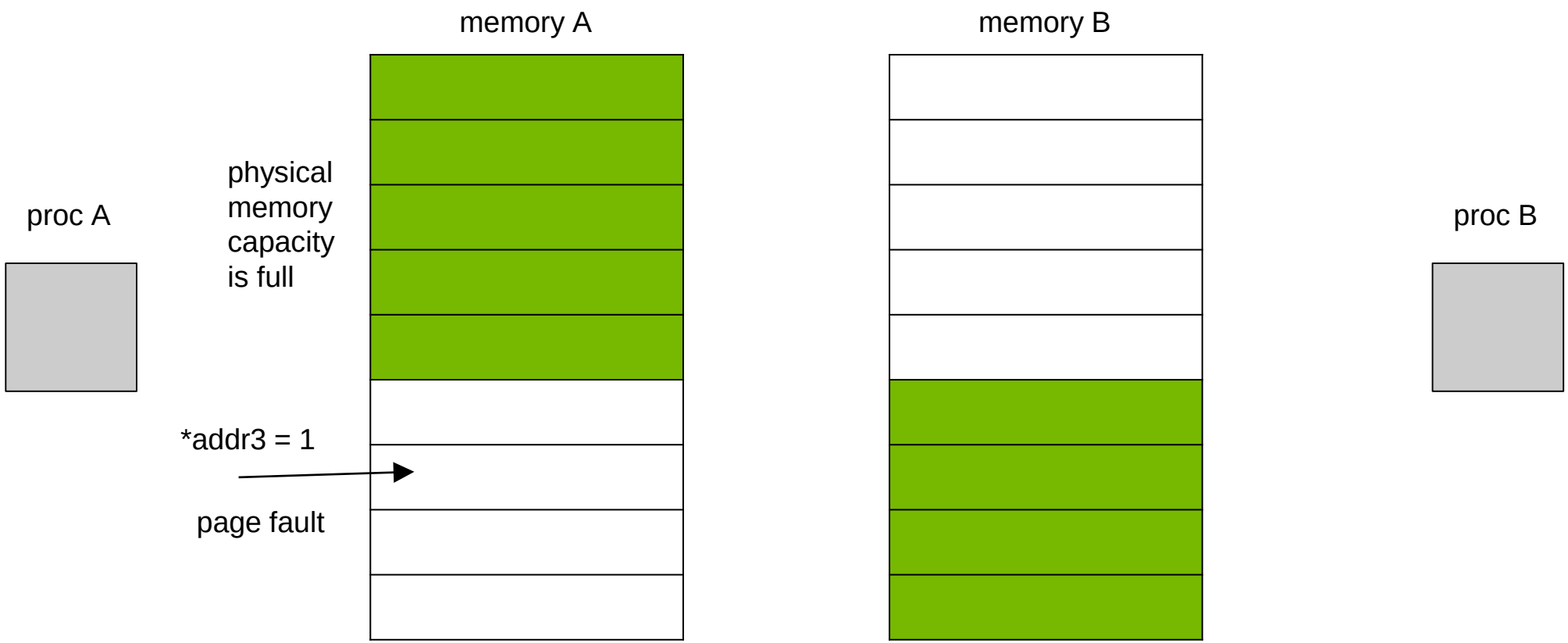
1) Quick and dirty algorithm prototyping
2) Iterative process with lots of data reuse, migration cost can be amortized
3) Simplify application debugging

When it's difficult to isolate the working set

1) Irregular or *dynamic* data structures, unpredictable access
2) Data partitioning between multiple processors

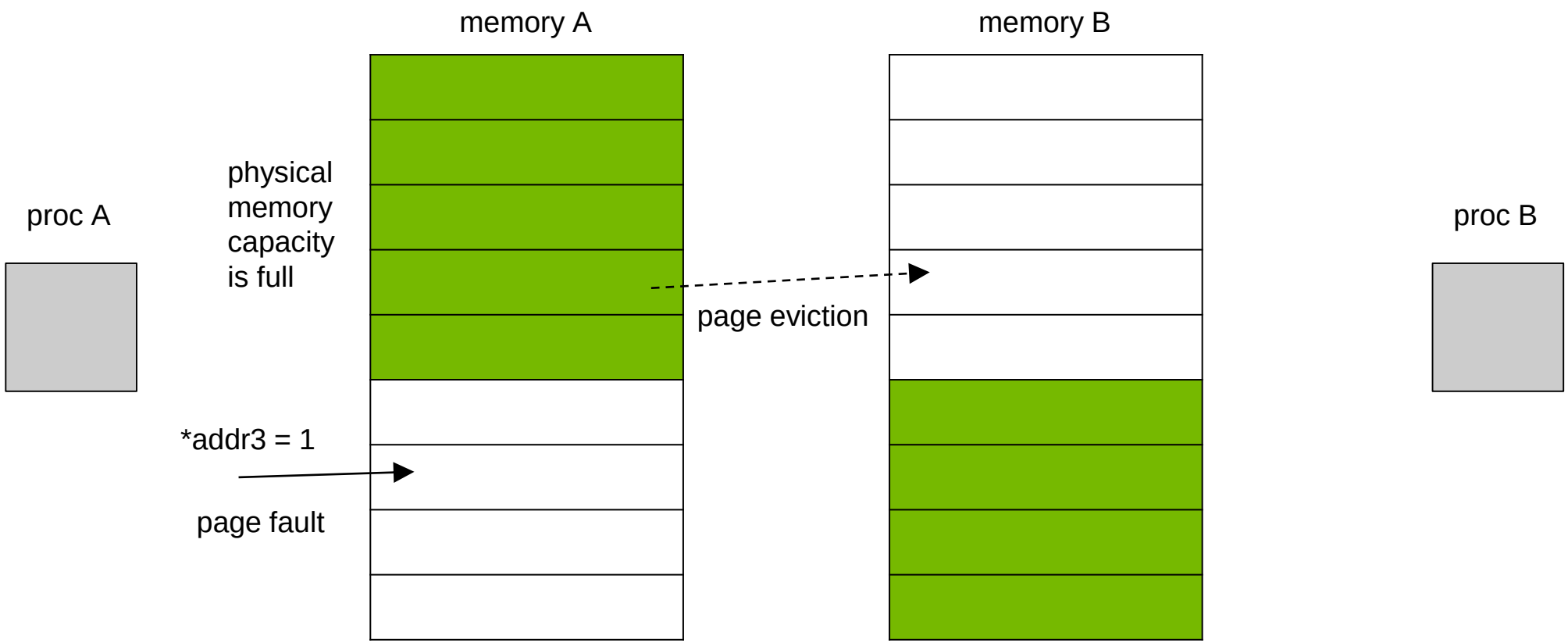
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



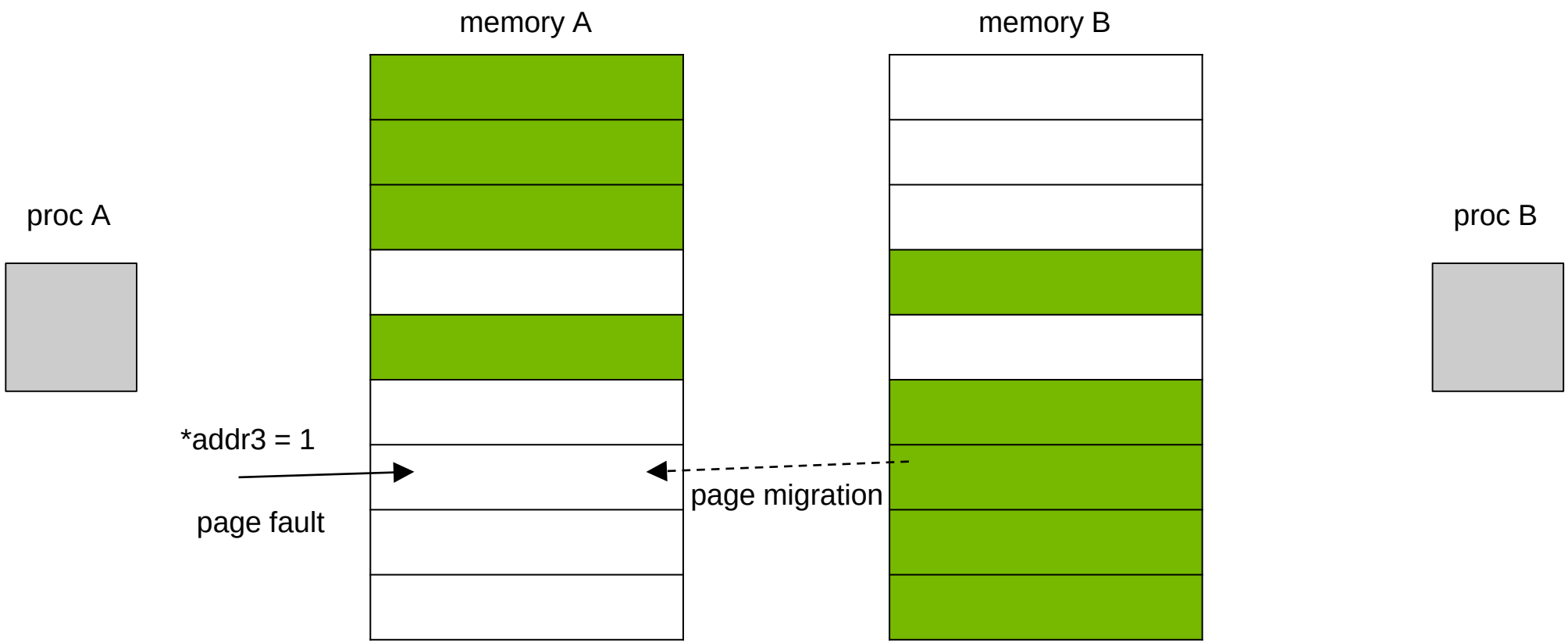
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



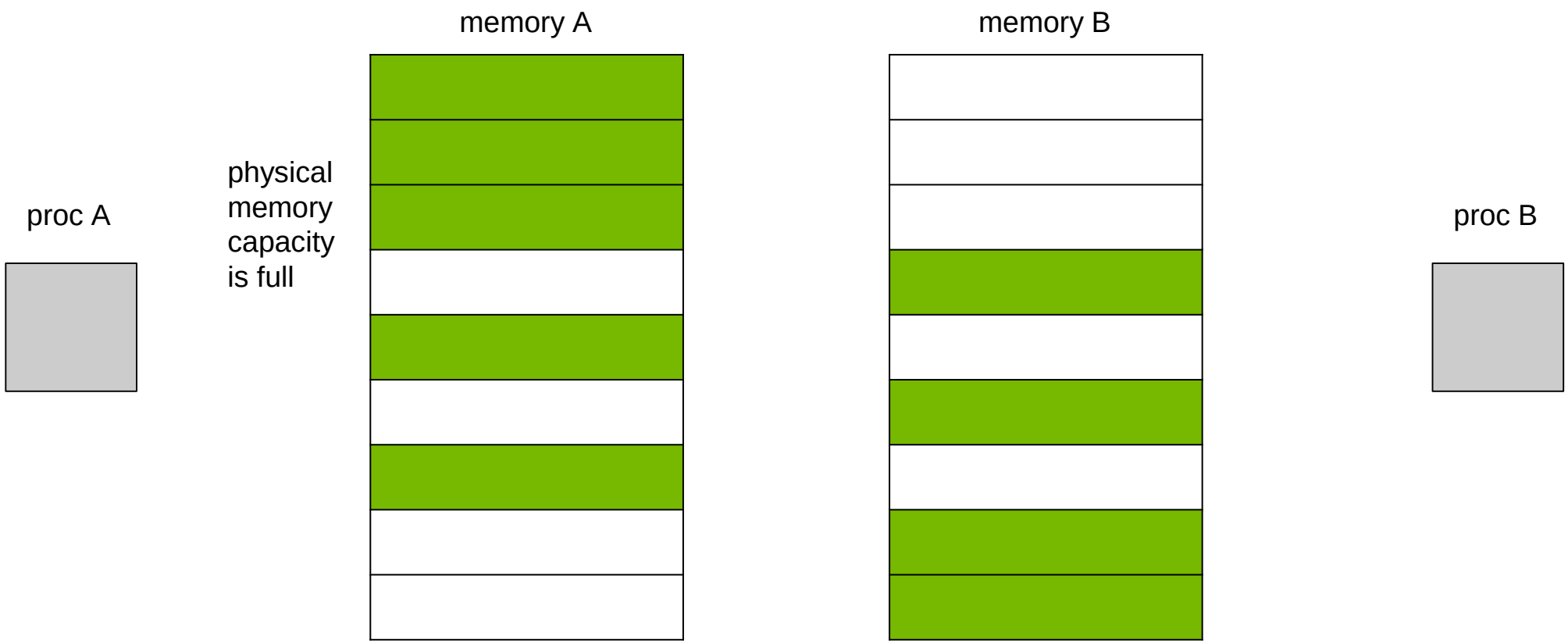
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription Benefits

When you have **large** dataset and not enough physical memory

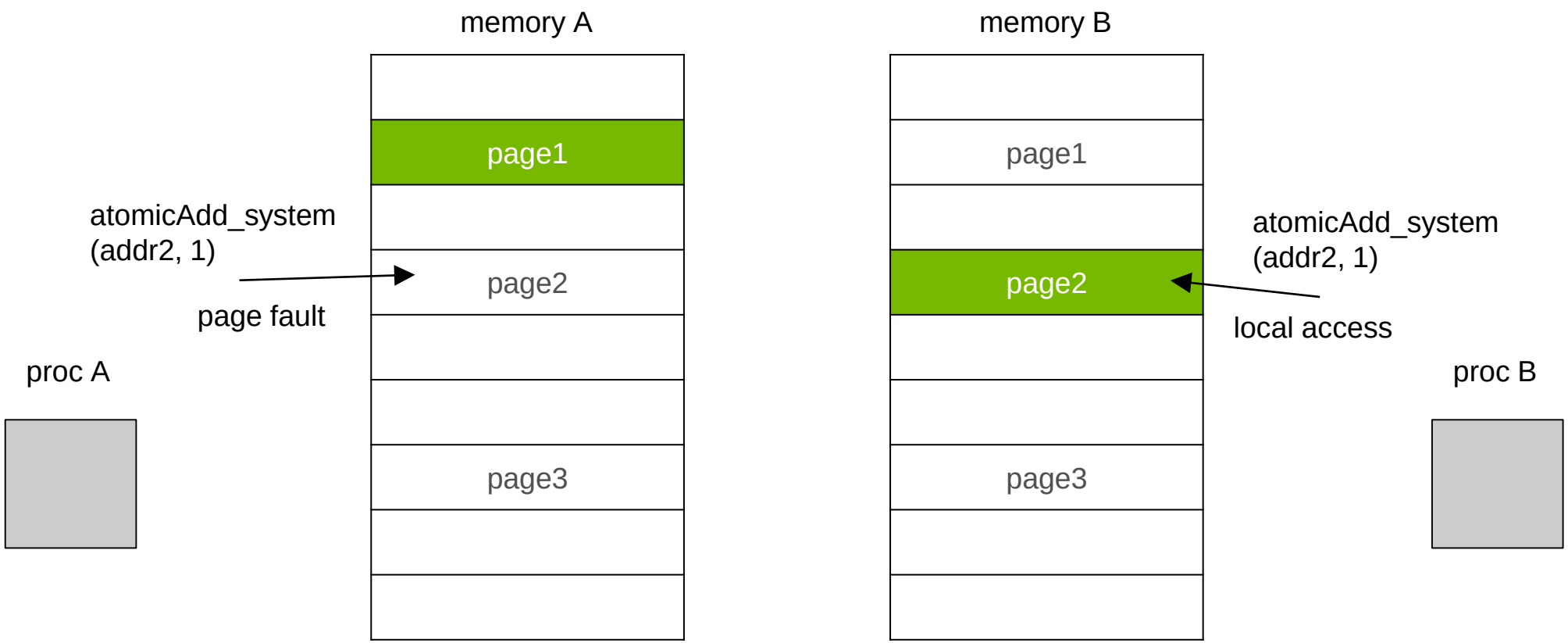
Moving pieces by hand is error-prone and requires tuning for memory size

Better to run slowly than get fail with out-of-memory error

You can actually get **high performance** with Unified Memory!

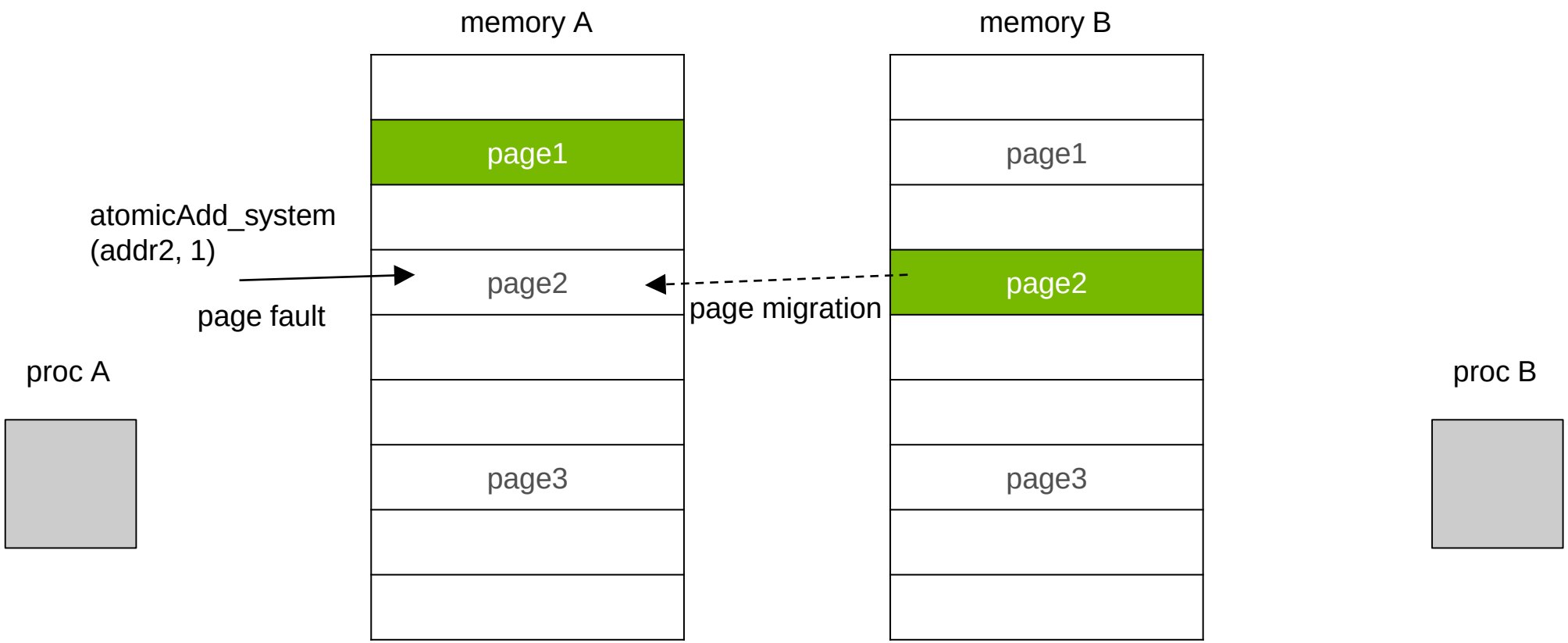
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



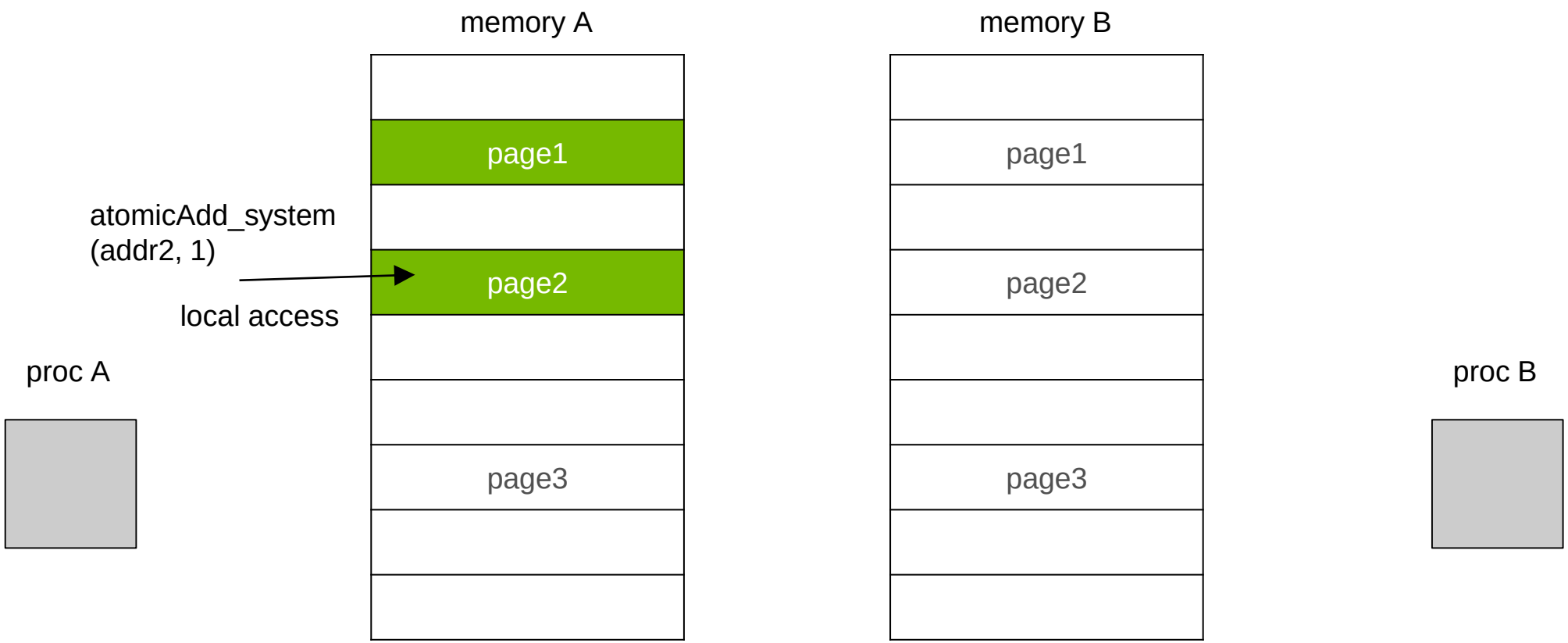
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



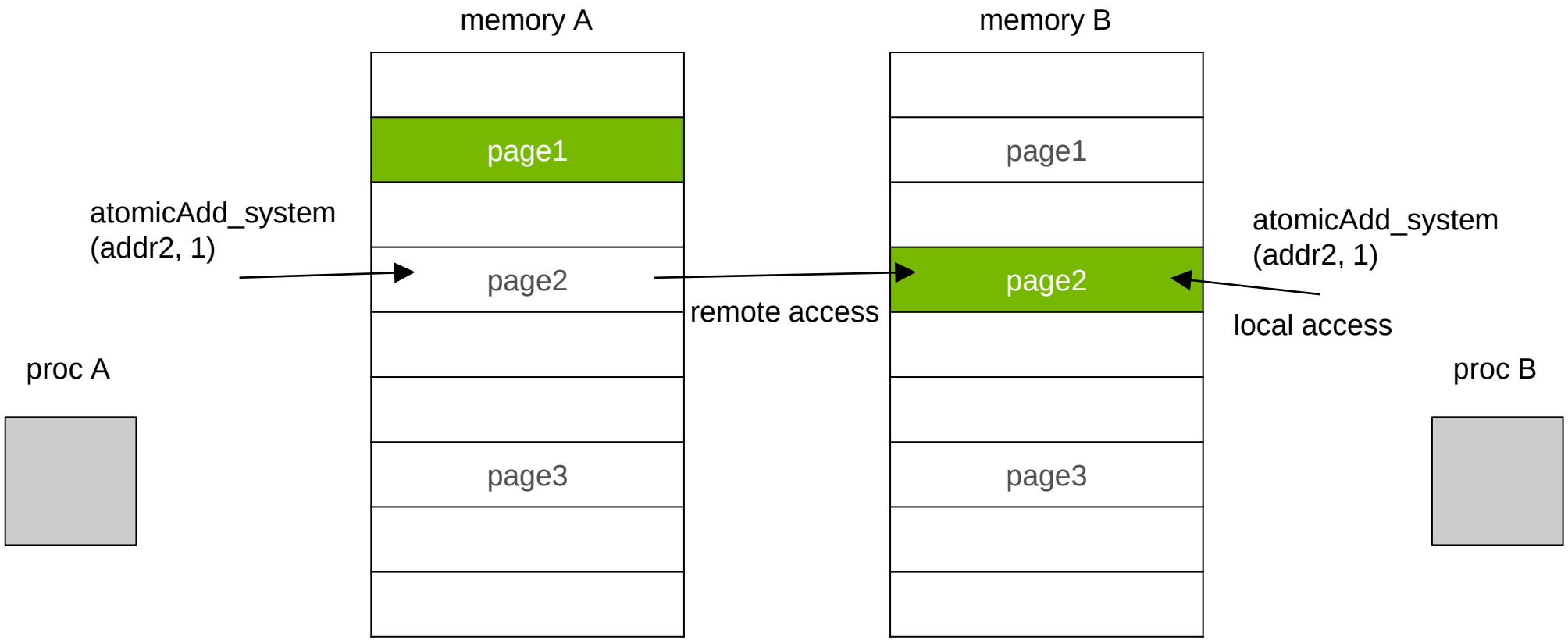
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics over NVLINK*



*both processors need to support atomic operations

UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics

GPUs are very good at handling atomics from *thousands of threads*

Makes sense to utilize atomics between GPUs or between CPU and GPU

We will see this in action on a realistic example later on

AGENDA

Unified Memory Fundamentals

Under the Hood Details

Performance Analysis and Optimizations

Applications Deep Dive

UNIFIED MEMORY ALLOCATOR

Available Options

CUDA C: **cudaMallocManaged** is your most reliable way to opt in today

CUDA Fortran: **managed** attribute (per allocation)

OpenACC: **-ta=managed** compiler option (all dynamic allocations)

malloc support is coming on Pascal+ architectures (Linux only)

Note: you can write your own malloc hook to use cudaMallocManaged

HETEROGEENOUS MEMORY MANAGER

Work In Progress

Heterogeneous Memory Manager: a set of Linux kernel patches

Allows GPUs to access all system memory (malloc, stack, file system)

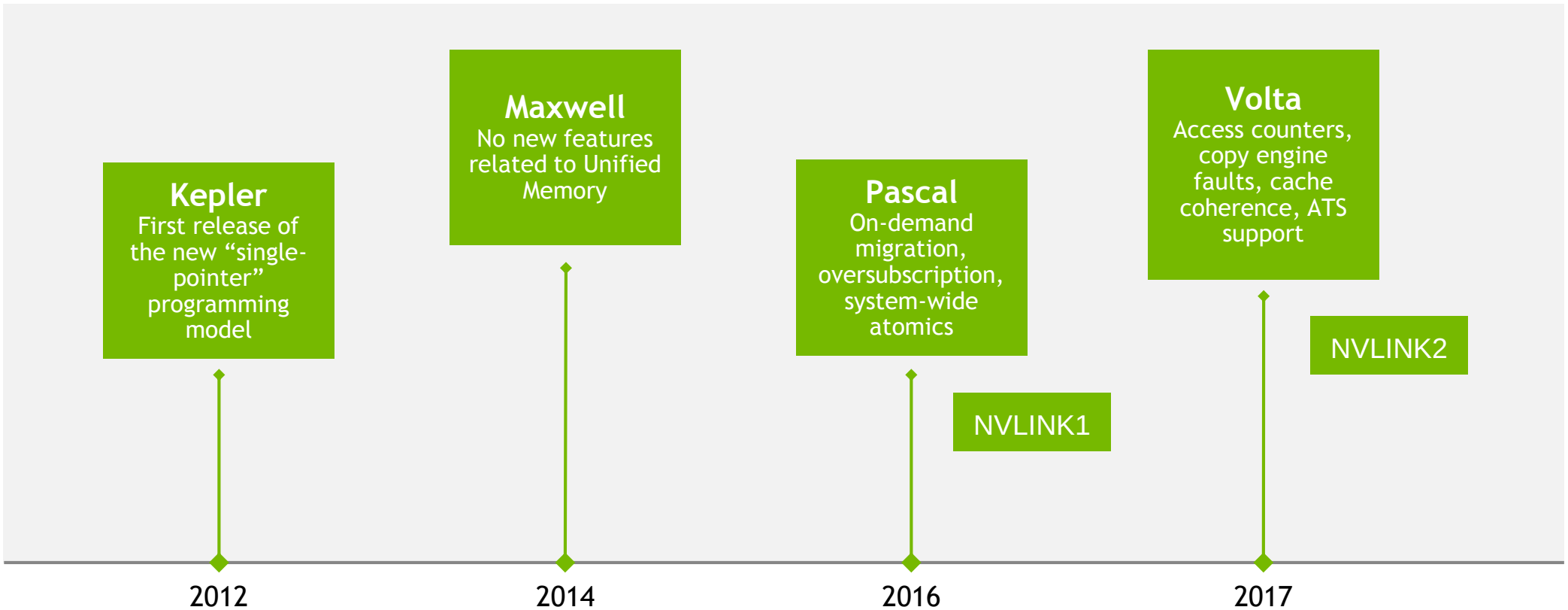
Page migration will be triggered the same way as for cudaMallocManaged

Ongoing testing and reviews, planning next phase of optimizations

More details on HMM today at 4:00 in Room 211B by John Hubbard

UNIFIED MEMORY

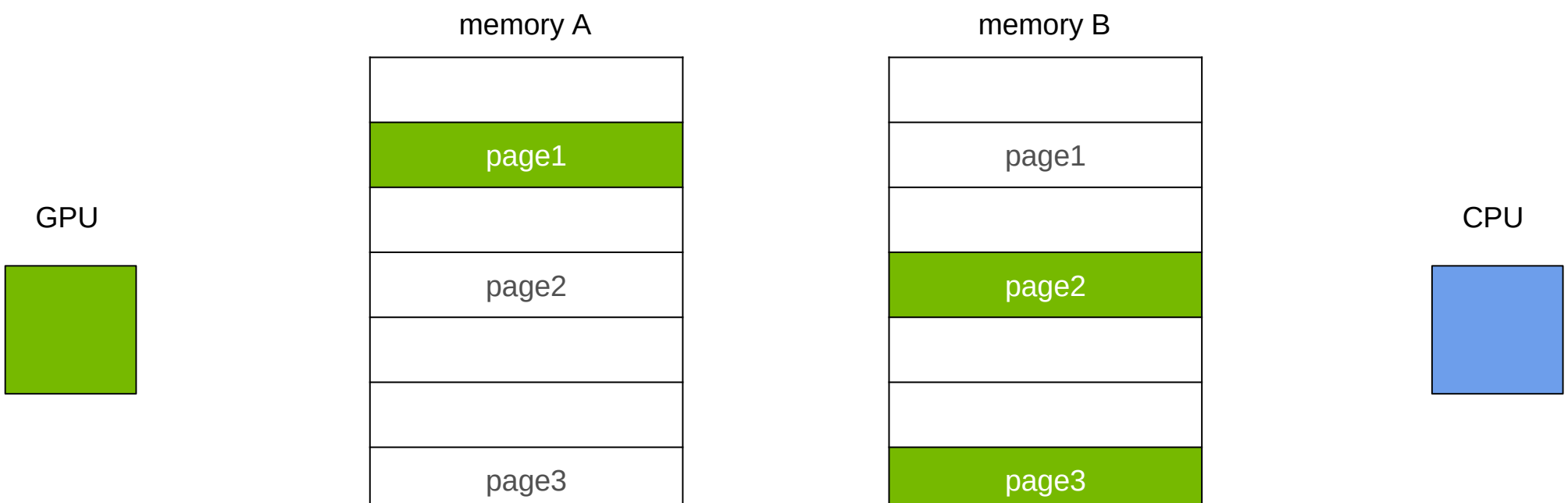
Evolution of GPU Architectures



UNIFIED MEMORY ON KEPLER

Available since CUDA 6

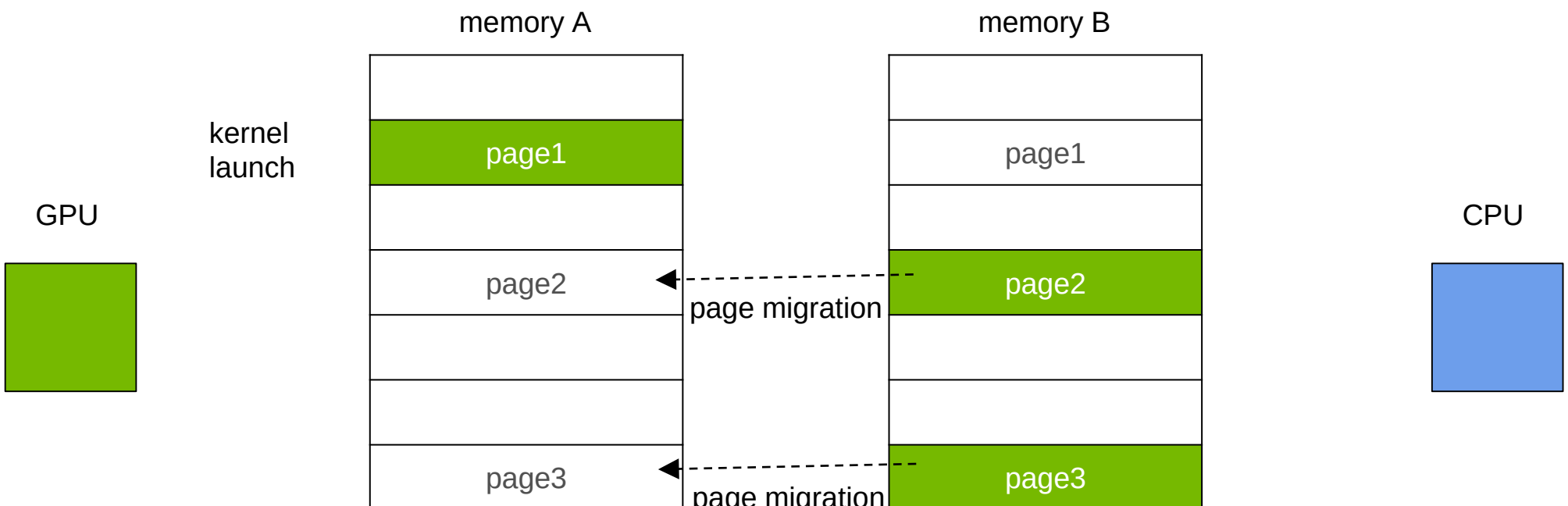
Kepler GPU: no page fault support, limited virtual space



UNIFIED MEMORY ON KEPLER

Available since CUDA 6

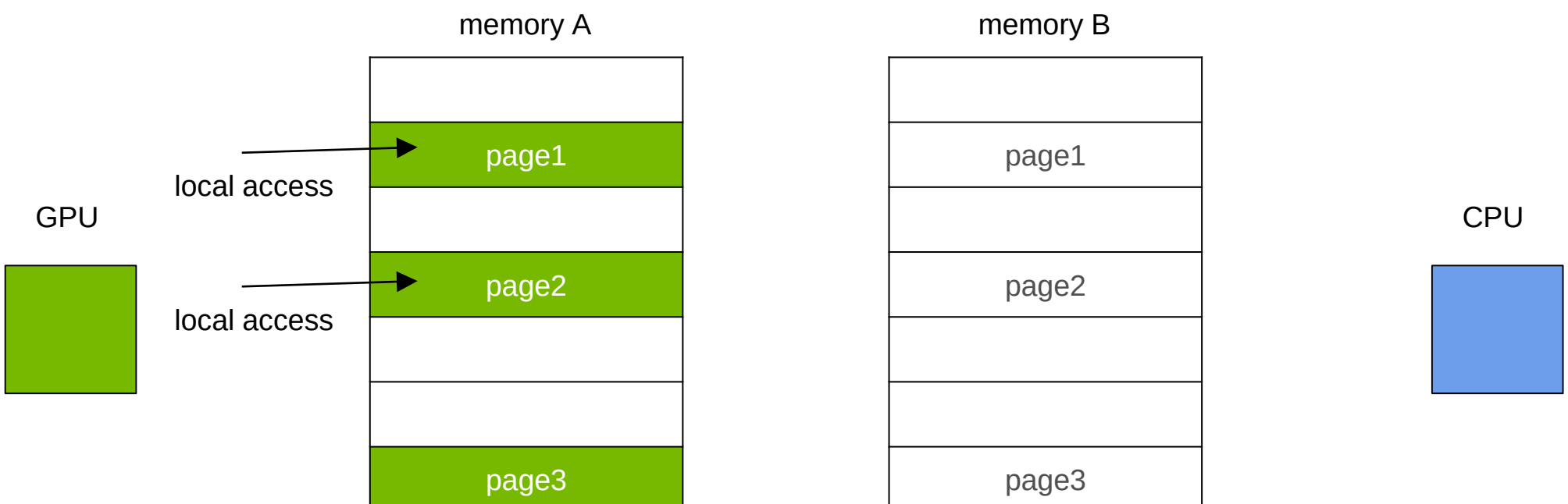
Bulk migration of **all pages** attached to current stream on kernel launch



UNIFIED MEMORY ON KEPLER

Available since CUDA 6

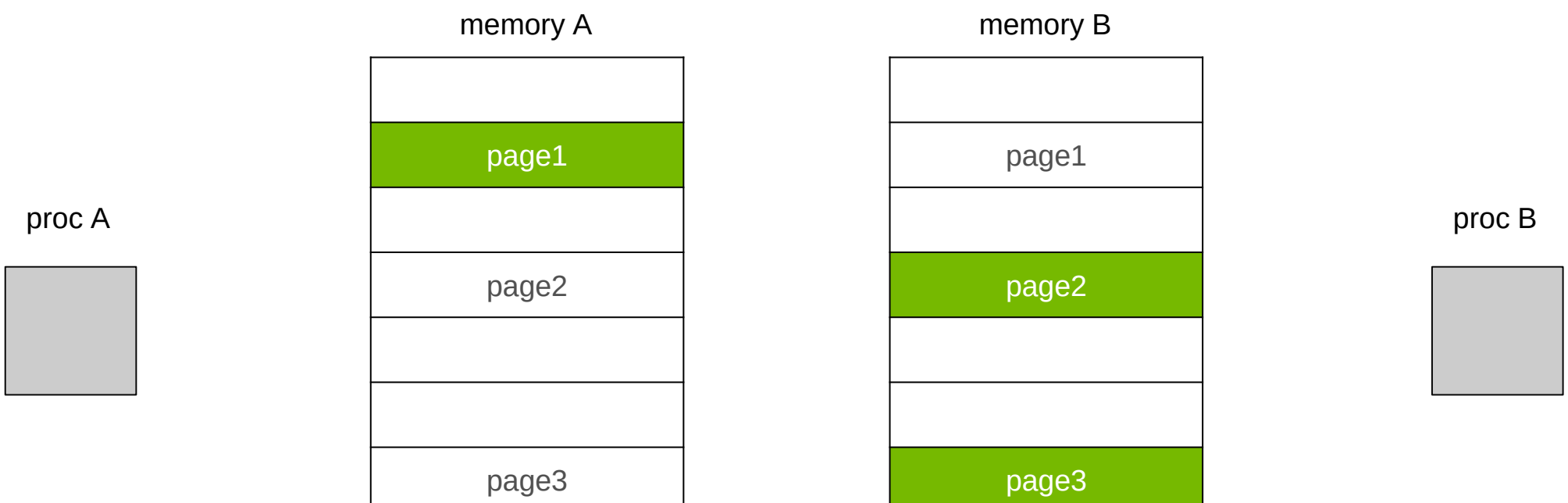
No on-demand migration for the GPU, no oversubscription, no system-wide atomics



UNIFIED MEMORY ON PASCAL

Available since CUDA 8

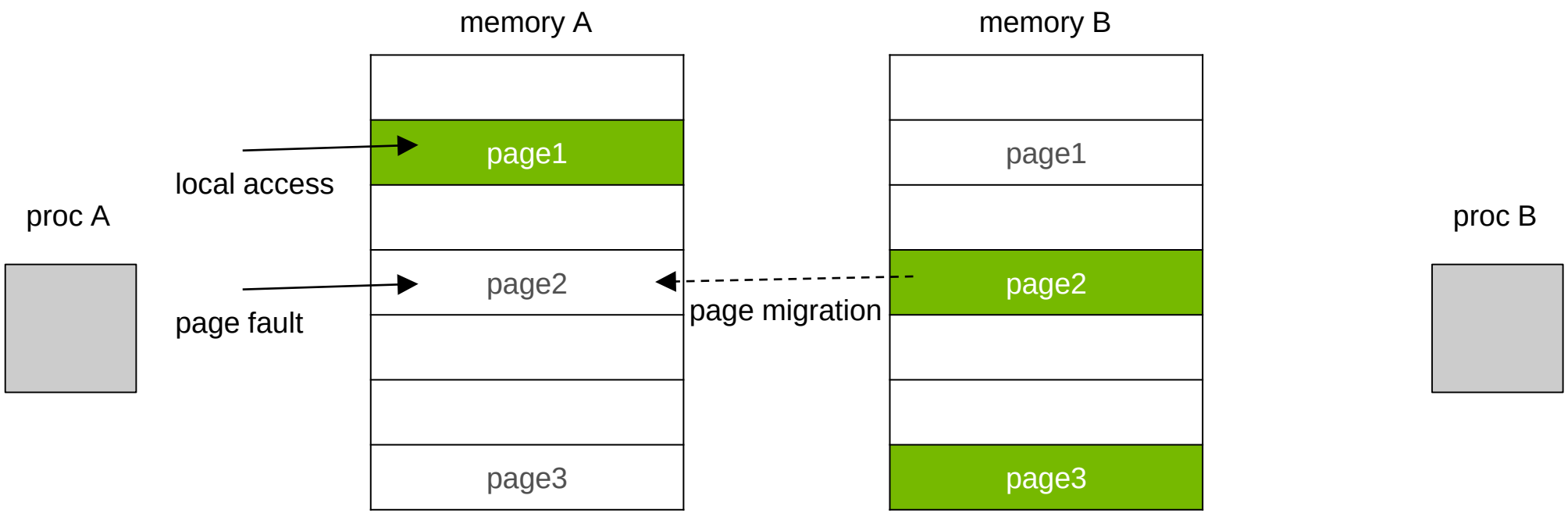
Pascal GPU: page fault support, extended virtual address space (48-bit)



UNIFIED MEMORY ON PASCAL

Available since CUDA 8

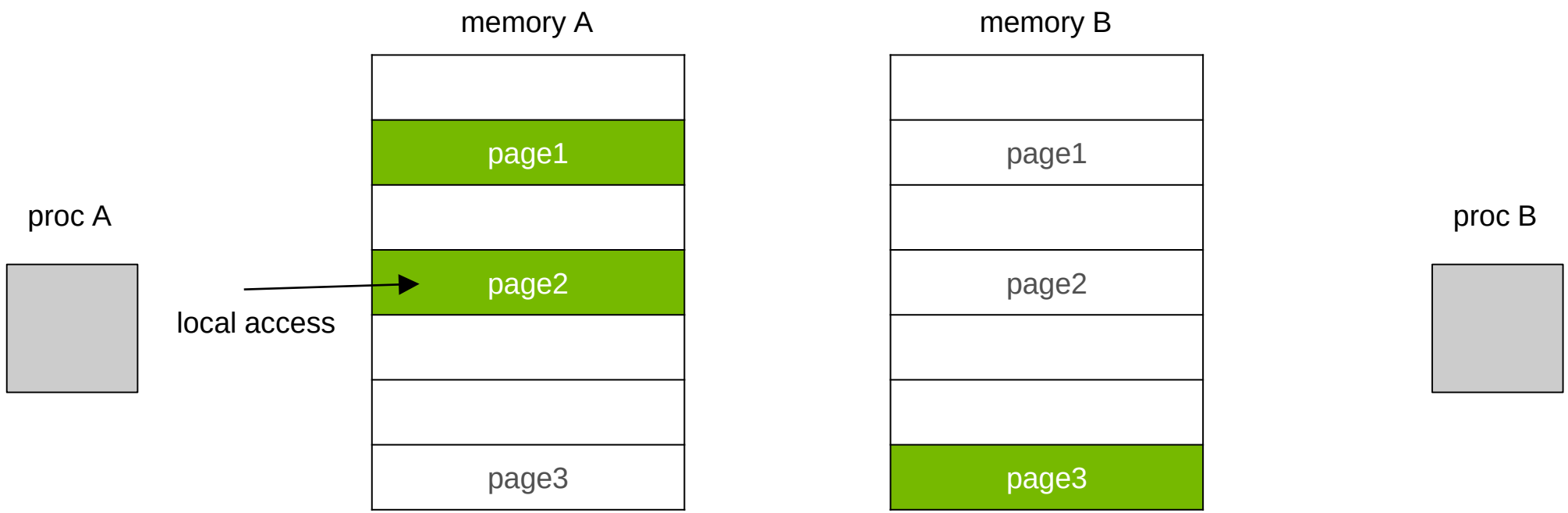
On-demand migration to accessing processor **on first touch**



UNIFIED MEMORY ON PASCAL

Available since CUDA 8

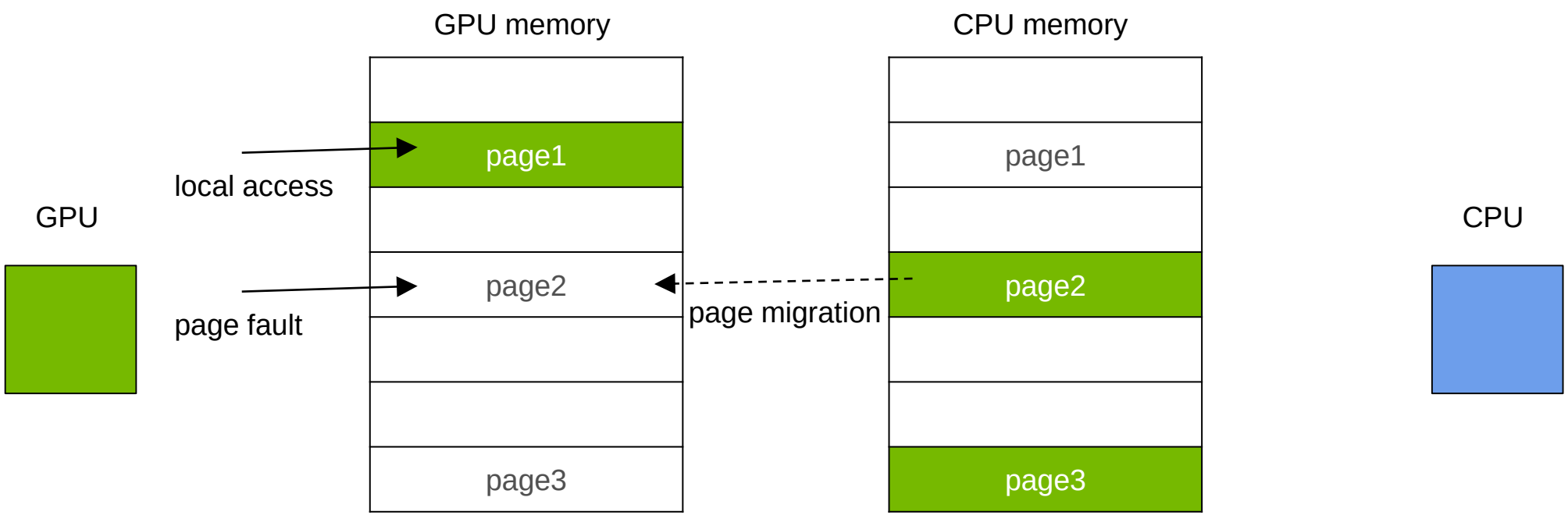
All features: on-demand migration, oversubscription, system-wide atomics



UNIFIED MEMORY ON VOLTA

Default model

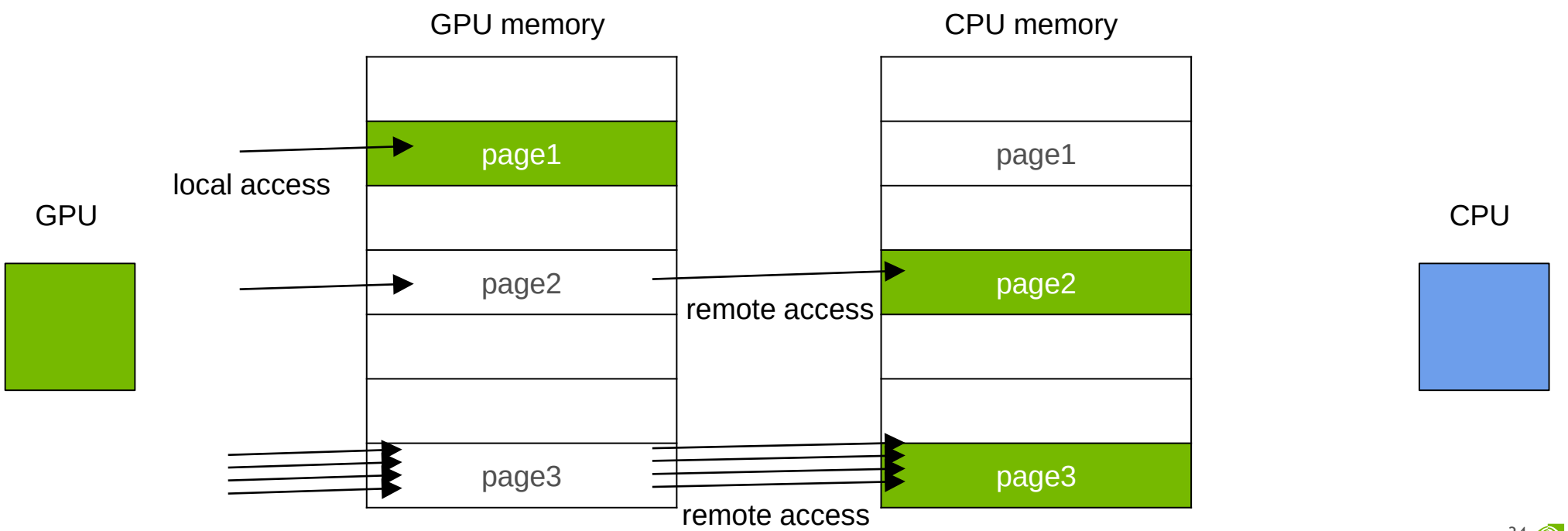
Volta GPU: uses fault on first touch for migration, **same as Pascal**



UNIFIED MEMORY ON VOLTA

New Feature: Access Counters

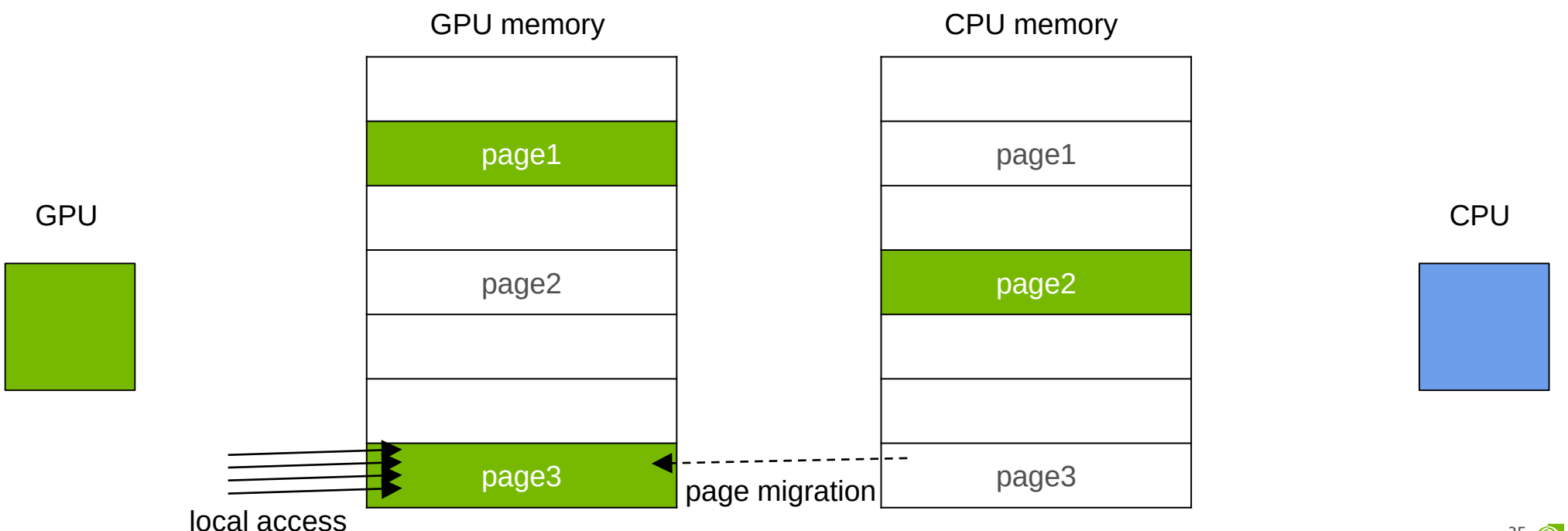
If memory is mapped to the GPU, migration can be triggered by access counters



UNIFIED MEMORY ON VOLTA

New Feature: Access Counters

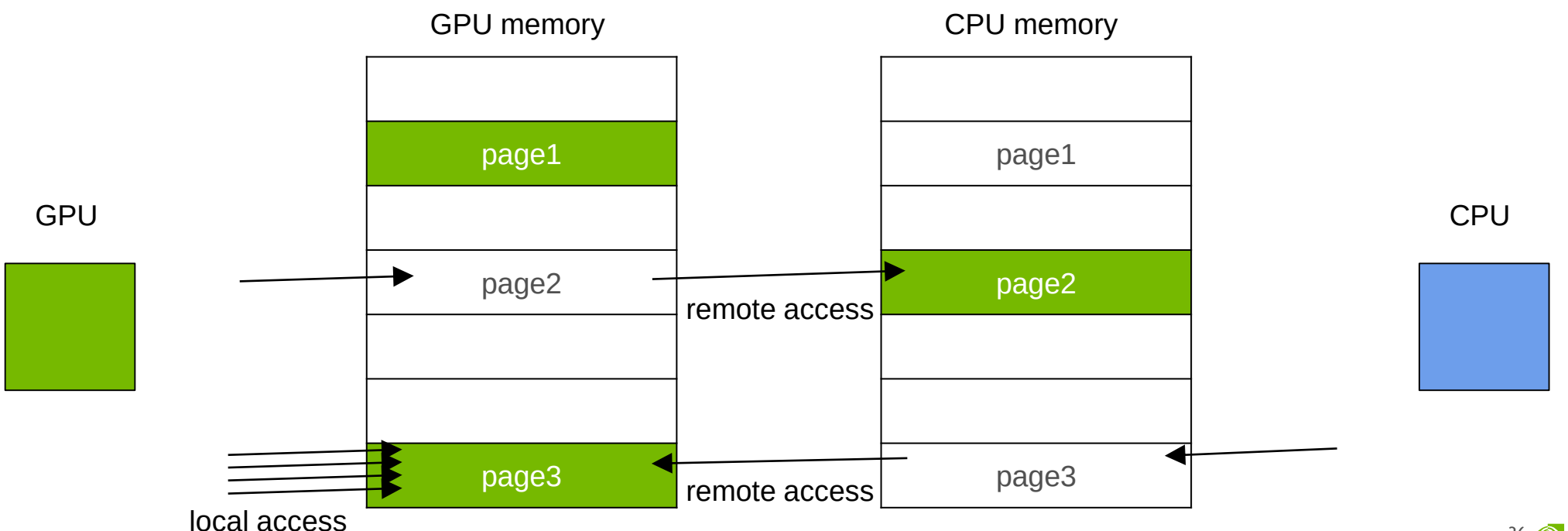
With access counters migration **only hot pages** will be moved to the GPU



UNIFIED MEMORY ON VOLTA+P9

NVLINK2: Cache Coherence

CPU can directly access and cache GPU memory; *native* CPU-GPU atomics



DRIVER HEURISTICS

Things You Didn't Know Exist

The Unified Memory driver is doing intelligent things under the hood:

Prefetching: migrate pages proactively to reduce number of faults

Thrashing mitigation: heuristics to avoid frequent migration of shared pages

Eviction: what pages to evict when we need to make the room for new ones

You can't control them but you can override most of these with hints

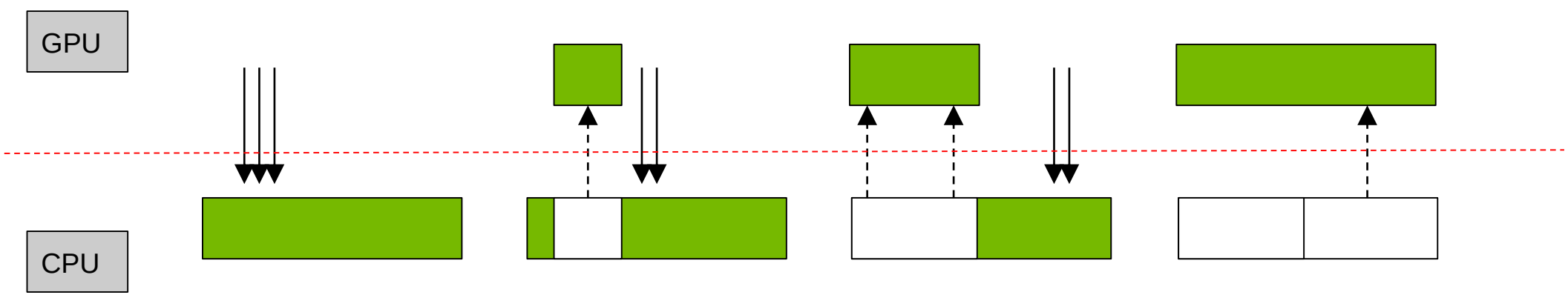
DRIVER PREFETCHING

Do Not Confuse with API-prefetching

GPU architecture supports different page sizes

Contiguous pages up to a larger page size are promoted to the larger size

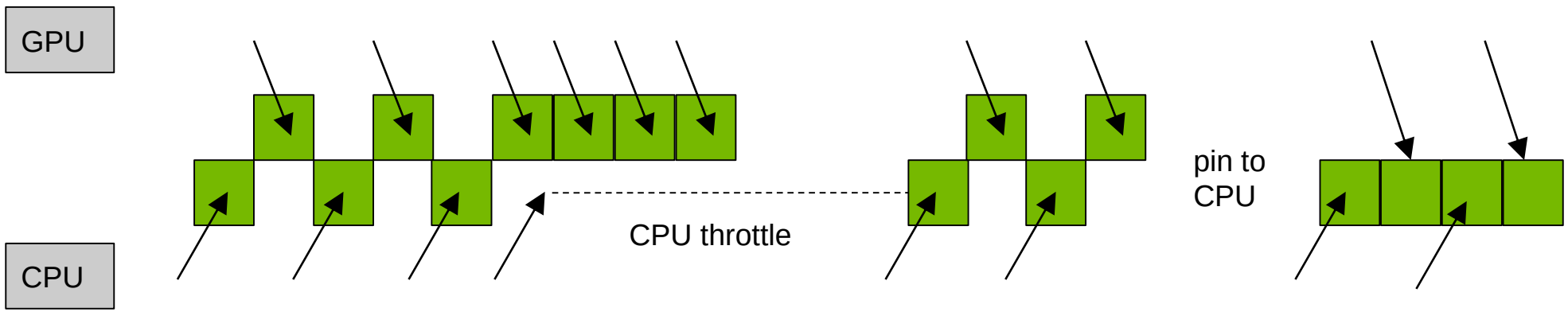
Driver prefetches whole regions if pages are accessed *densely*



ANTI-THRASHING POLICY

Frequent Access to Shared Data

Processors share the same page and frequently read or write to it



Pascal: when memory is pinned we lose any insight into access pattern

Volta: can use access counters information to find a better location

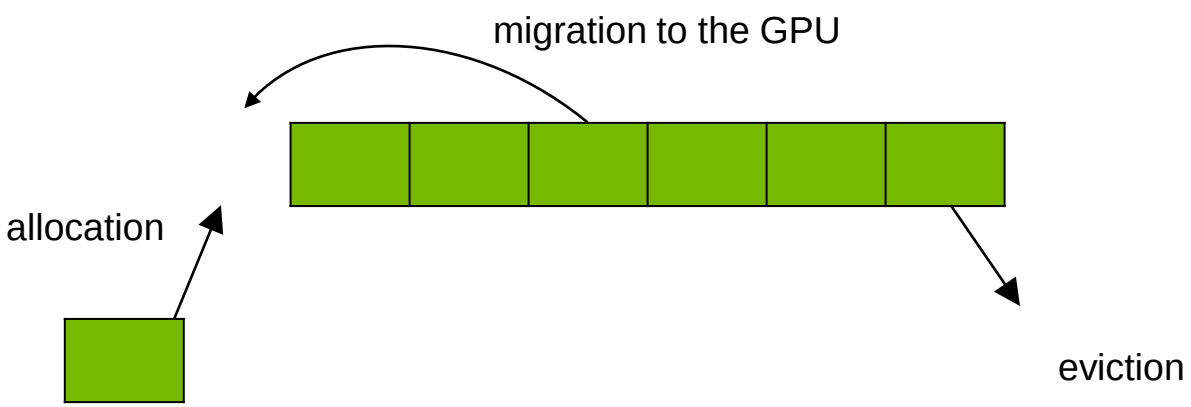
EVICTION ALGORITHM

What Pages Are Moving Out of the GPU

Driver keeps a single list of physical chunks of GPU memory

Chunks from the front of the list are evicted first (LRU)

A chunk is considered “in use” when it is fully-populated or migrated



AGENDA

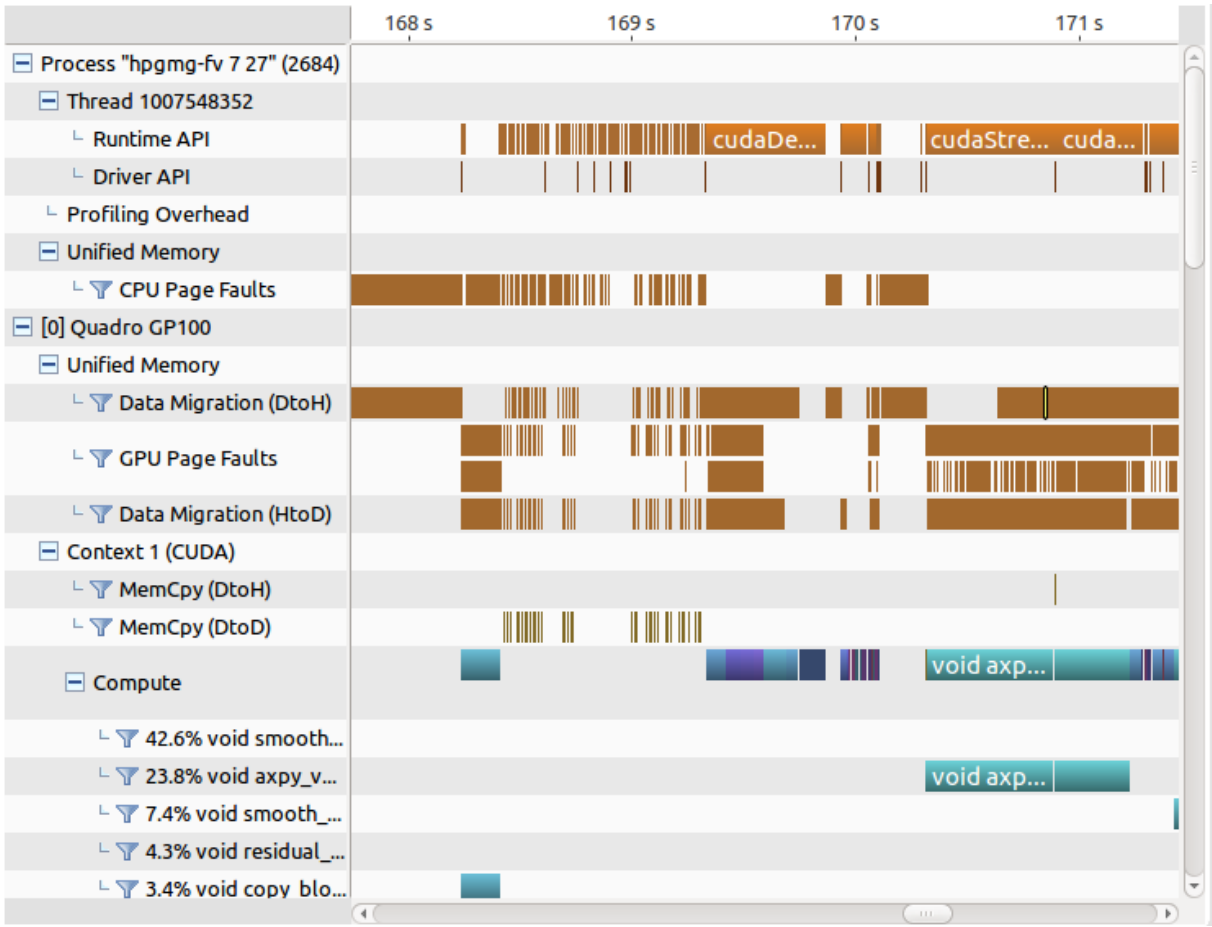
Unified Memory Fundamentals

Under the Hood Details

Performance Analysis and Optimizations

Applications Deep Dive

PROFILER: INSPECT



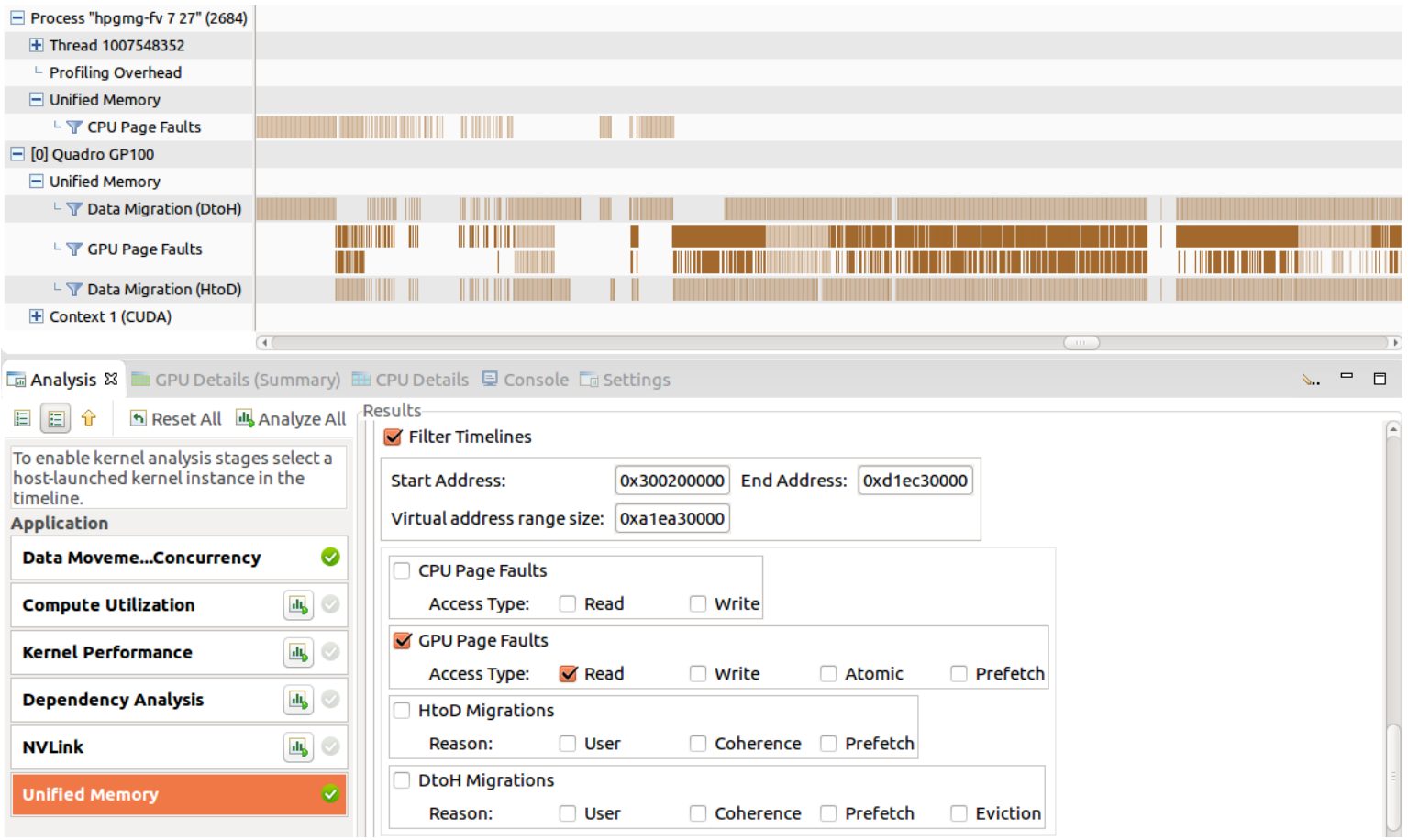
Data Migration (DtoH)

Start	170.845 s (170,844,511,496 ns)
End	170.845 s (170,844,671,976 ns)
Duration	160.48 µs
Size	2.097 MB
Migration Cause	Eviction
Throughput	13.068 GB/s
Virtual Address	0x4b2a00000
Memory operations	Browse for executable...
Process	2684

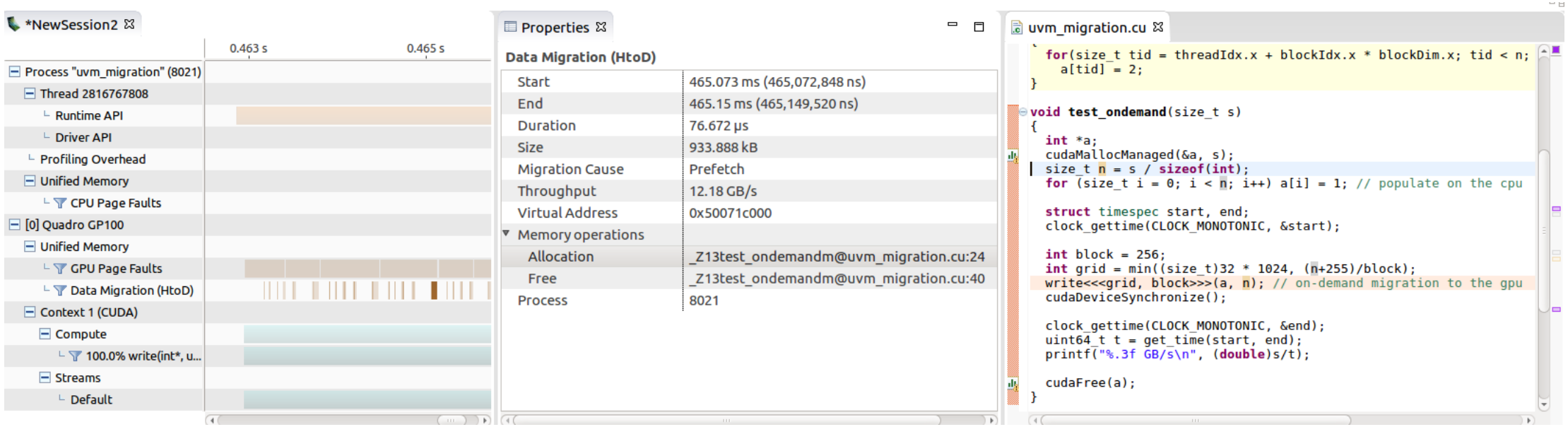
GPU Page Fault groups

Start	170.925 s (170,924,798,600 ns)
End	170.925 s (170,924,919,048 ns)
Duration	120.448 µs
Memory Acccess Type	Write
Virtual Address	0x34b680000
GPU Page Faults	2
Memory operations	Browse for executable...
Process	2684

PROFILER: FILTER



PROFILER: CORRELATE



More details tomorrow at 10:00 in Marriott Salon 3

USER HINTS

Why, When, and How to Use Them

If you know your application well you can optimize with hints

These are also useful to override some of the driver heuristics

`cudaMemPrefetchAsync``(ptr, size, processor, stream)`

Similar to `move_pages()` in Linux

`cudaMemAdvise``(ptr, size, advice, processor)`

Similar to `madvise()` in Linux

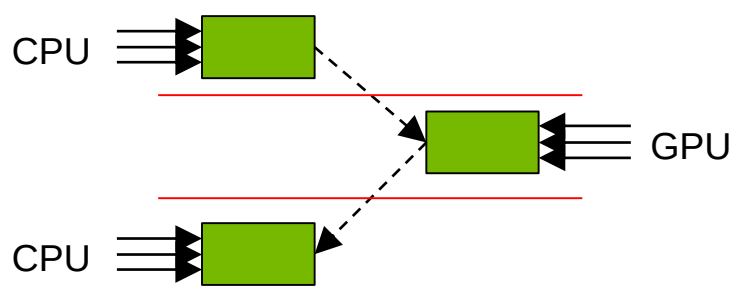
USER HINTS

Prefetching

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemPrefetchAsync(data, N, myGpuId, s);
mykernel<<<..., s>>>(data, N);
cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s);
cudaStreamSynchronize(s);

use_data(data, N);

cudaFree(data);
```

Page faults can be expensive and they stall SM execution

Avoid faults by prefetching data to the accessing processor



USER HINTS

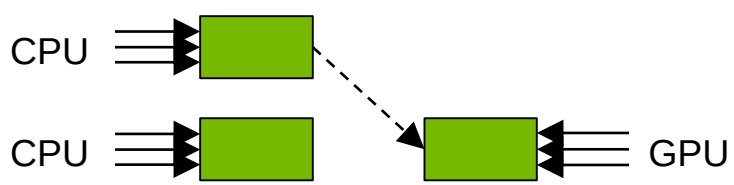
Read Mostly

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetReadMostly, myGpuId);
cudaMemPrefetchAsync(data, N, myGpuId, s);
mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

In this case prefetch creates a copy instead of moving data

Both processors can read data **simultaneously** without faults

Writes are allowed but they are expensive



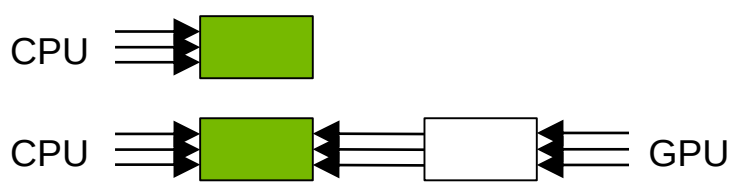
USER HINTS

Preferred Location

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

Here the kernel will *page fault* and generate direct mapping to data on the CPU

The driver will “resist” migrating data away from the preferred location



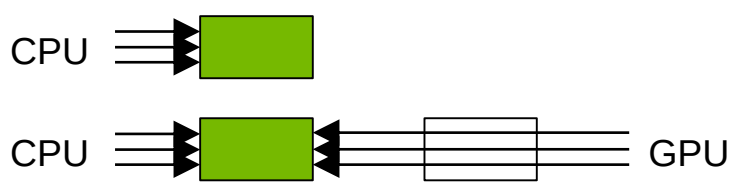
USER HINTS

Accessed By

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

Memory can move freely to other processors and mapping will carry over



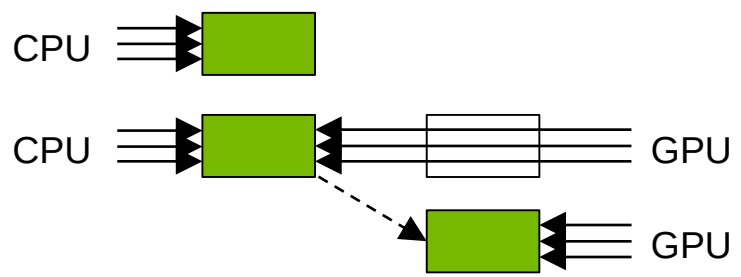
USER HINTS

Accessed By on Volta

```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);

mykernel<<<..., s>>>(data, N);


use_data(data, N);

cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

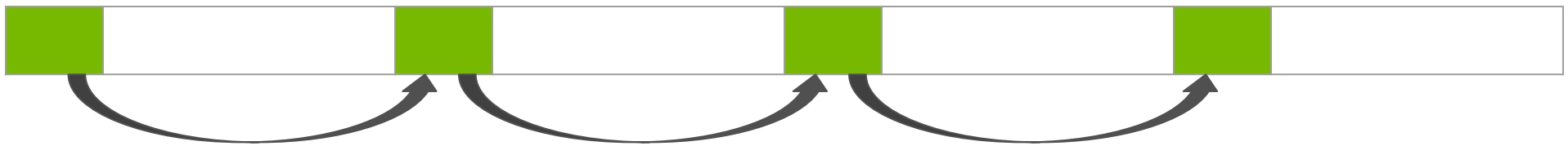
Access counters may eventually trigger migration of this memory to the GPU



PERFORMANCE

Page Fault Cost

How long does a page fault take to serve? - We can measure!



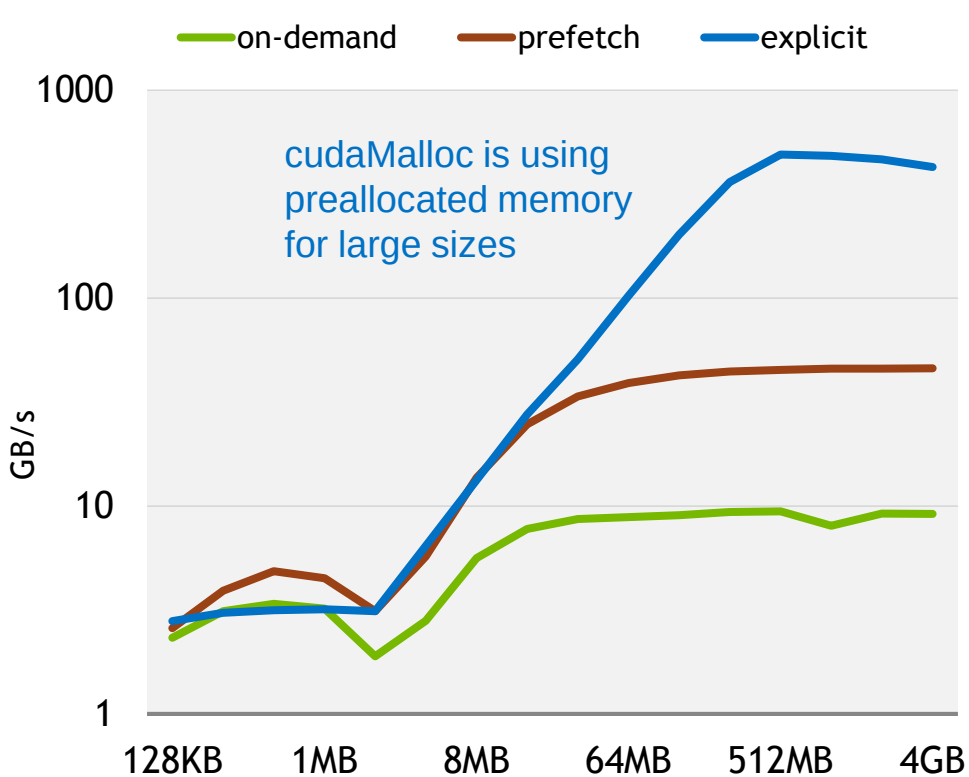
Linked list traversal with some large stride to avoid prefetching effects

Page fault cost (us)	DtoH	HtoD
x86 + PCIe + GP100	20	30
P8 + NVLINK + GP100	20	20

PERFORMANCE

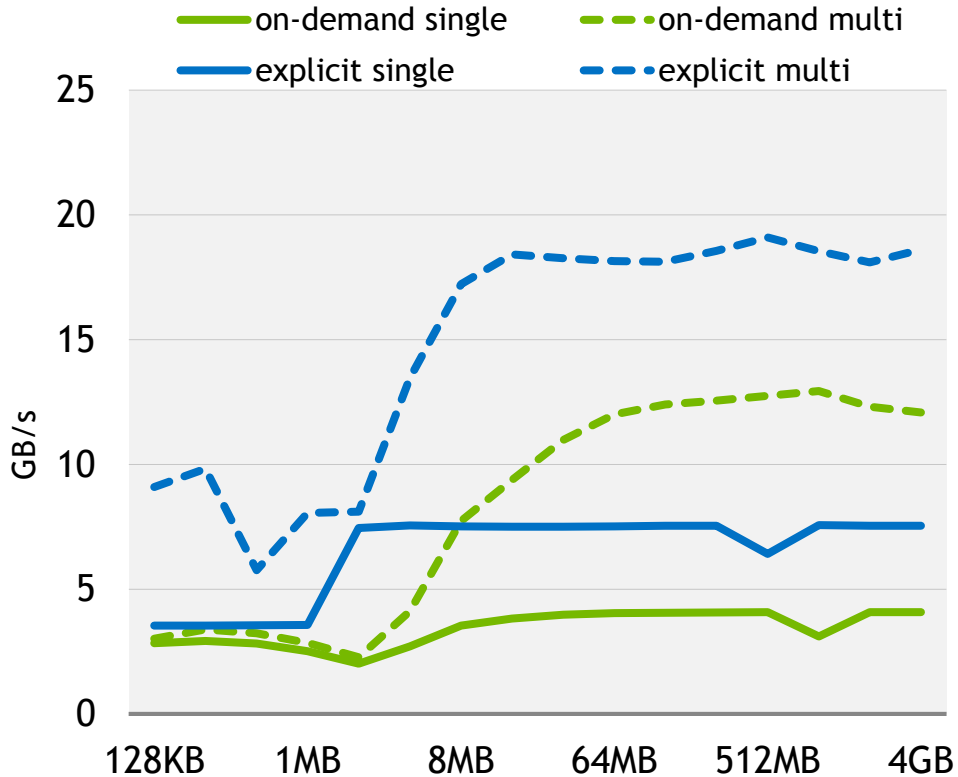
Page Allocation Throughput

GPU memory



cudaMallocManaged/cudaMalloc + cudaMemset

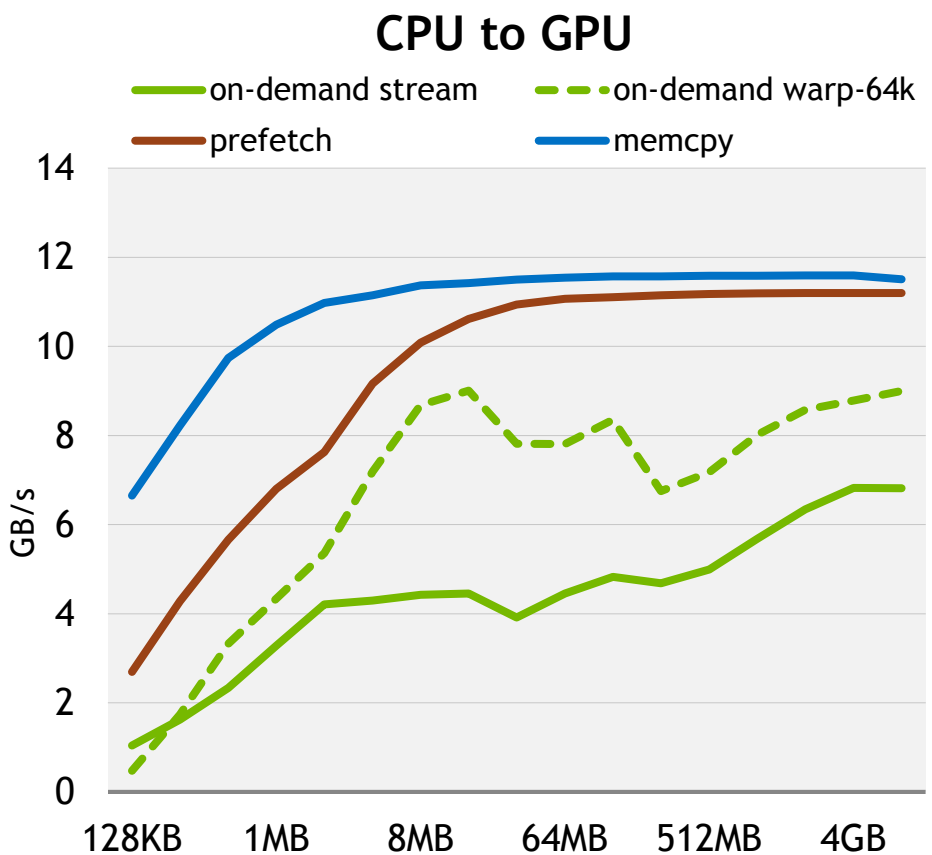
CPU memory

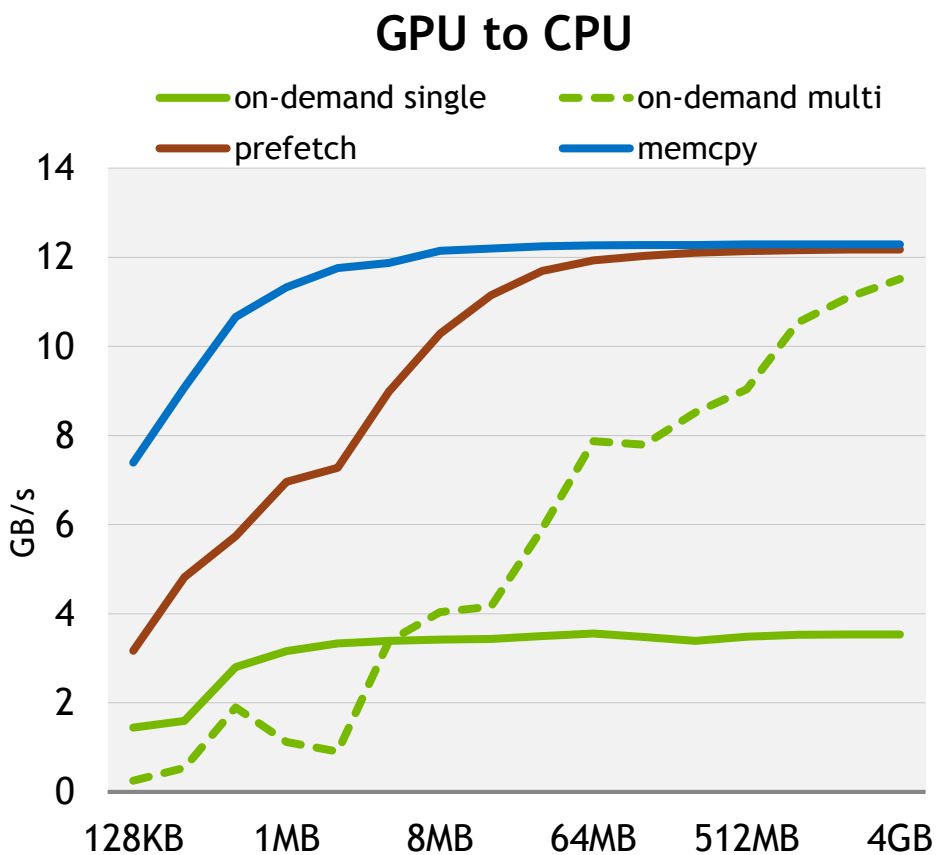


cudaMallocManaged/mmap + fill on the CPU

PERFORMANCE

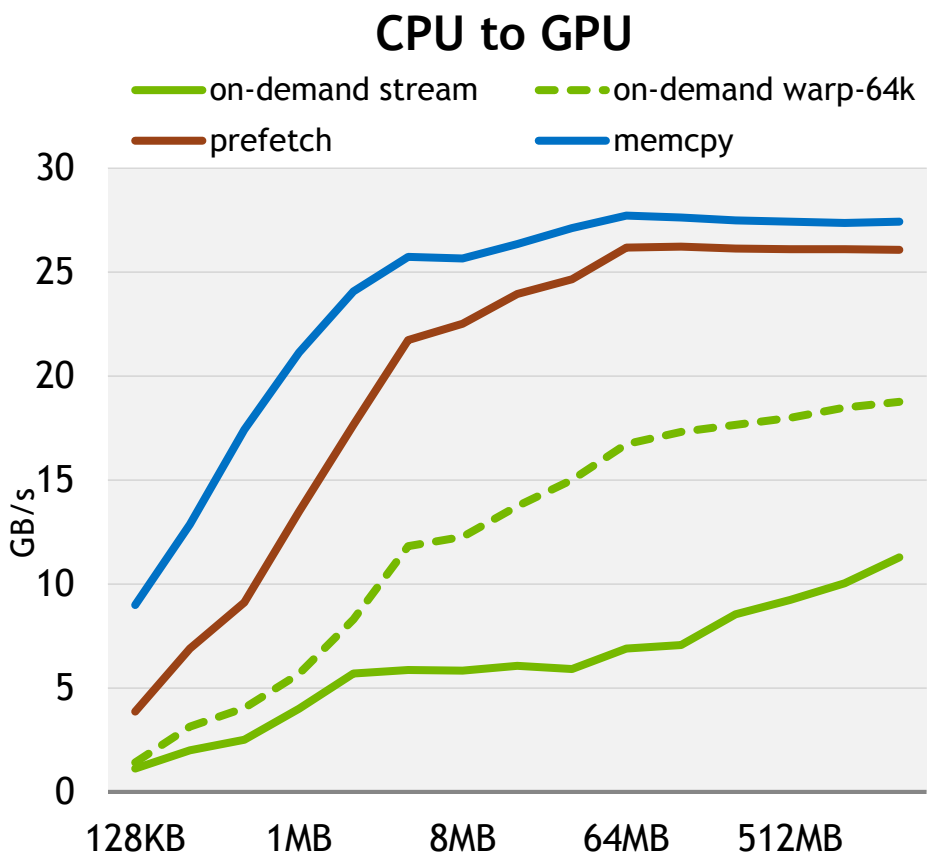
Page Migration Throughput (PCIe)

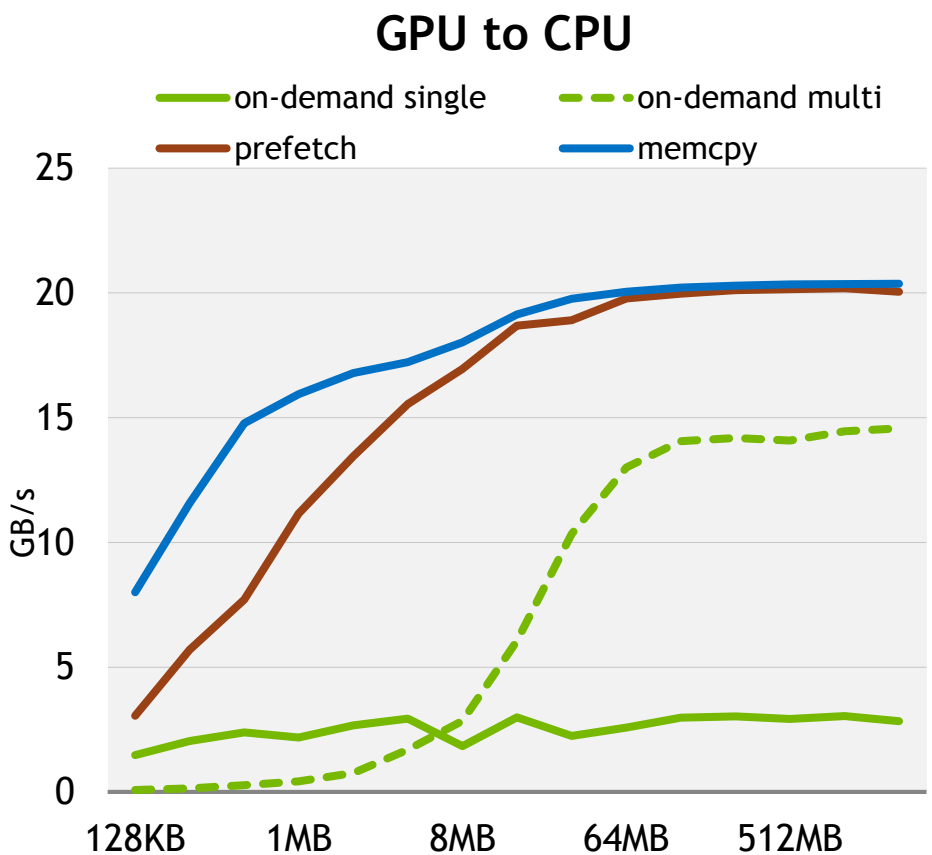




PERFORMANCE

Page Migration Throughput (2x NVLINK)





PERFORMANCE

Page Granularity Overhead

cudaMallocManaged *alignment*: 512B on Pascal/Volta, 4KB on Kepler/Maxwell

Too many small allocations will use up many pages

cudaMallocManaged memory is moved at *system page* granularity

For small allocations more data could be moved than necessary

Solution: use cached allocator or memory pools

AGENDA

Unified Memory Fundamentals

Under the Hood Details

Performance Analysis and Optimizations

Applications Deep Dive

HPC: HPGMG

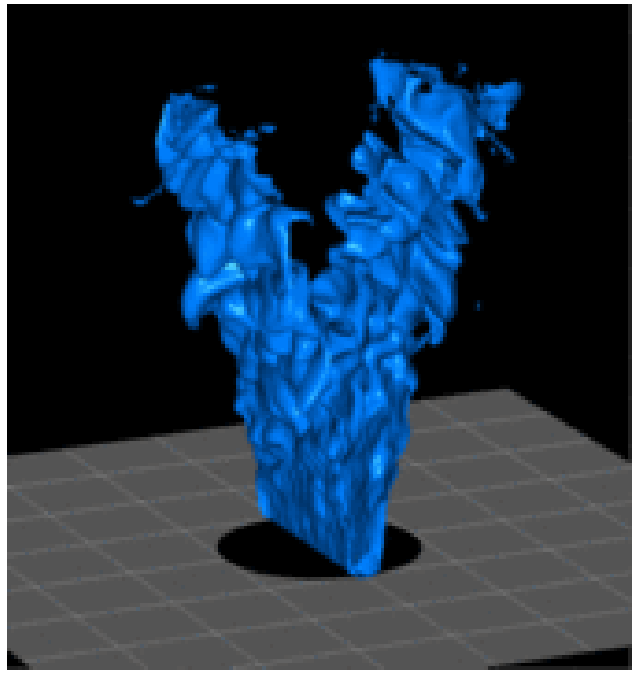
Combustion Simulation

High-Performance Geometric Multigrid

Proxy AMR and Low Mach Combustion codes

Used in Top500 benchmarking

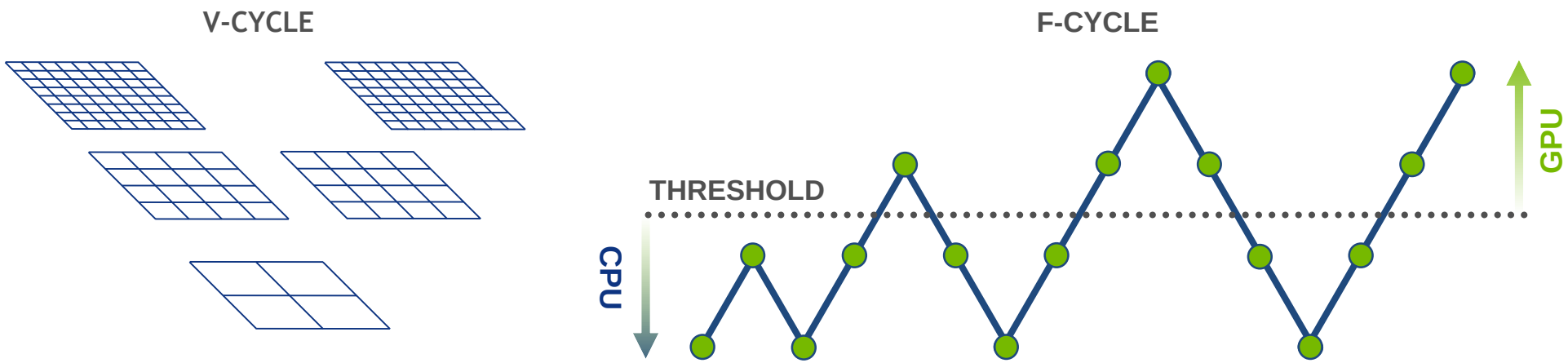
High memory usage requirements



http://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg/

HPC: HPGMG

Taking Advantage of the CPU and the GPU

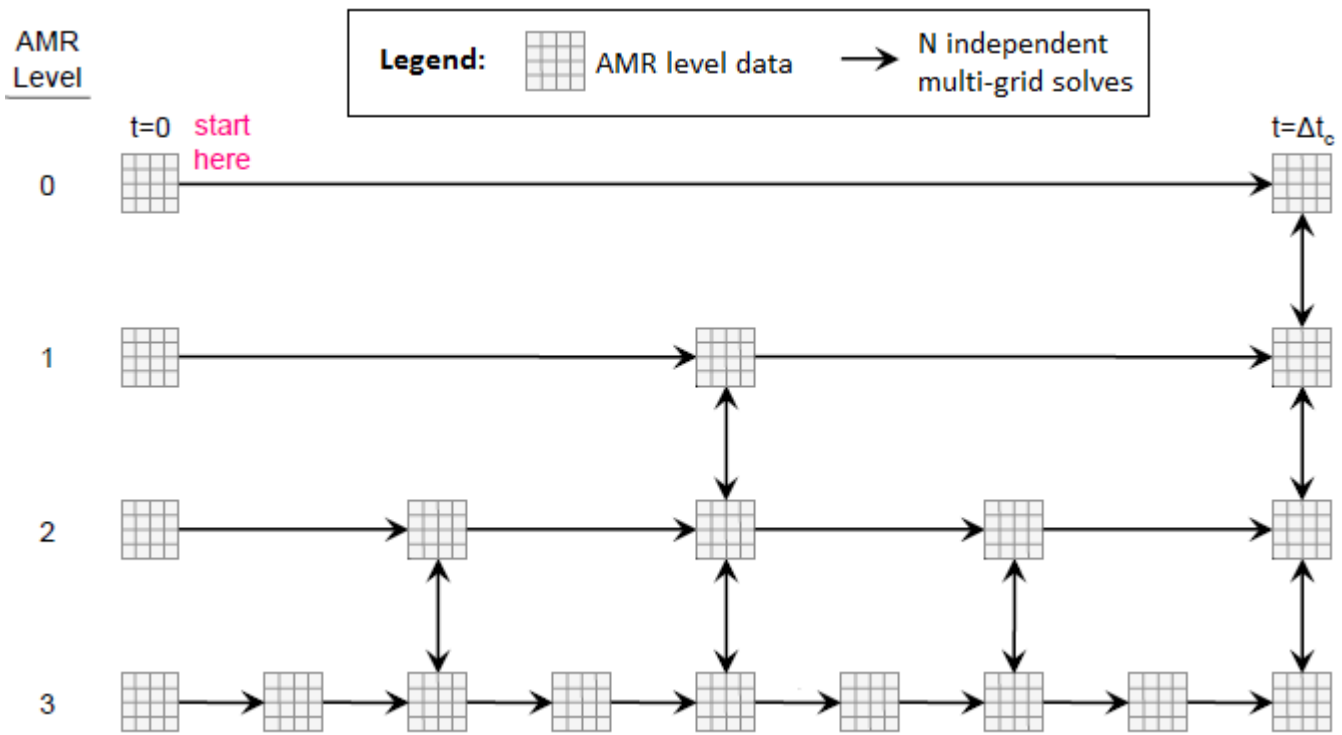


Hybrid implementation requires very careful memory management

Frequent data sharing when crossing the CPU-GPU threshold

HPGMG: AMR PROXY

Data Locality and Reuse of AMR Levels

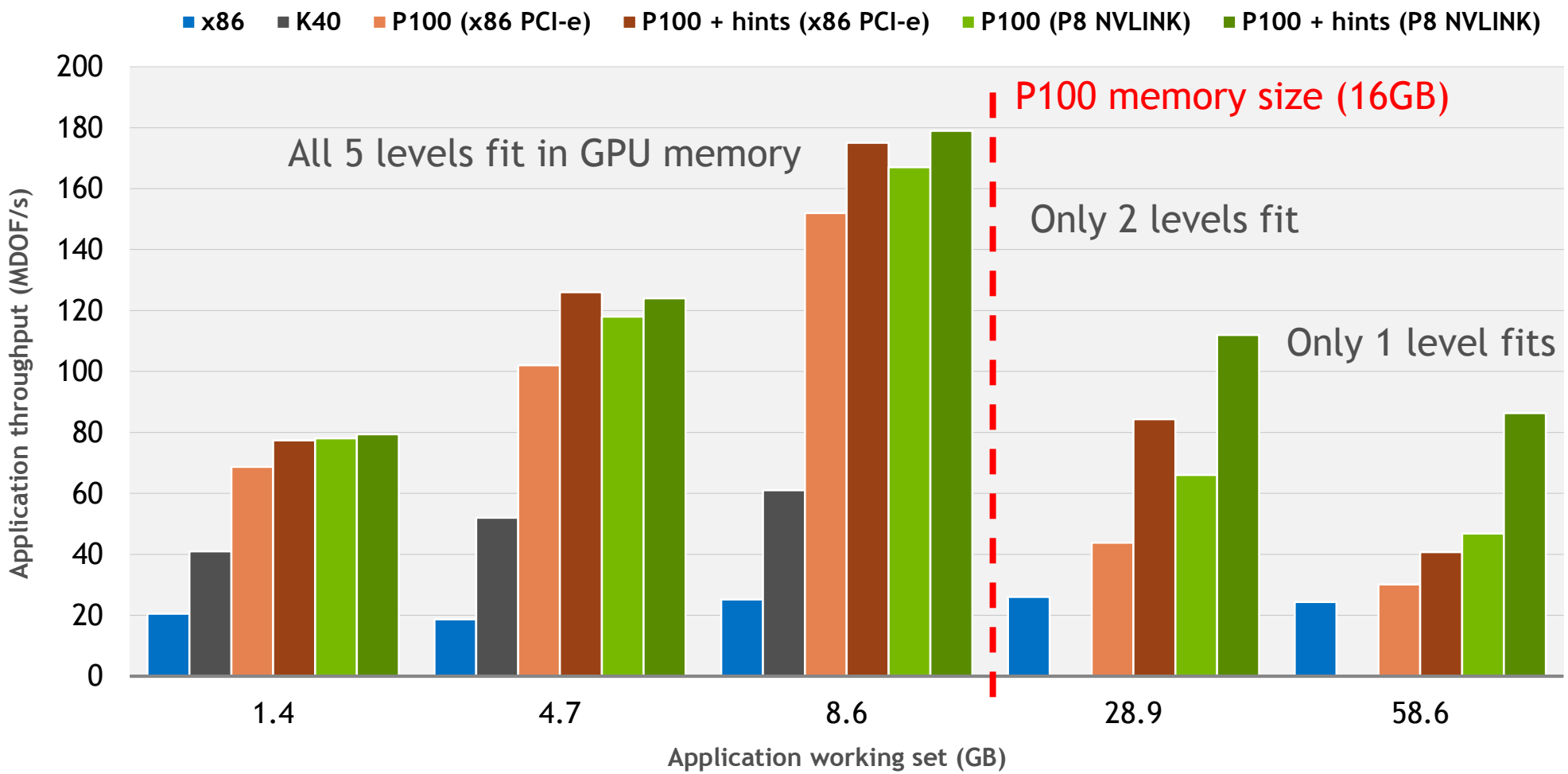


Optimization: prefetch the next AMR level while running computations on the current level

We can use a separate non-blocking CUDA stream to overlap with the default stream

https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/

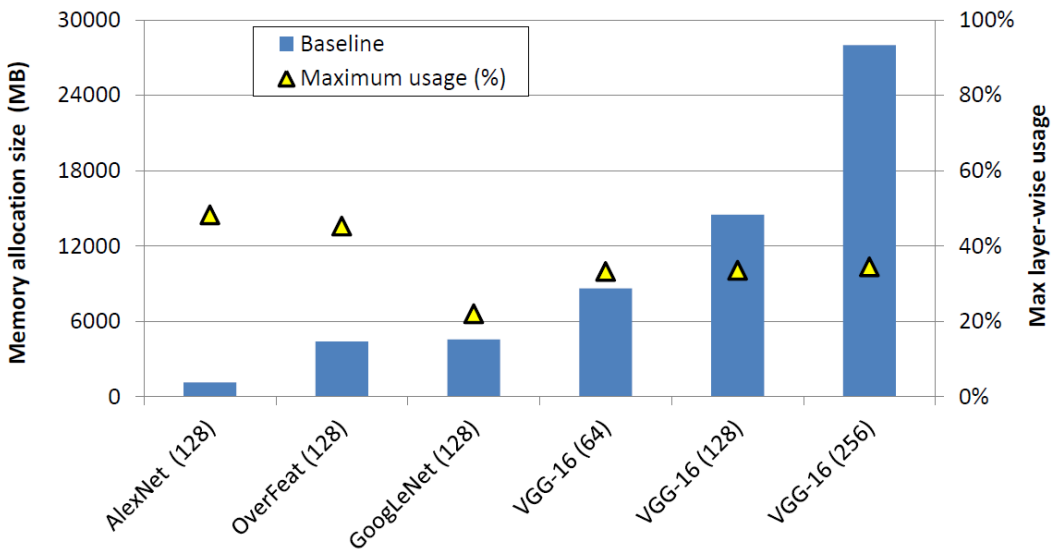
AMR PROXY OVERSUBSCRIPTION



***x86 CPU**: Intel E5-2630 v3, 2 sockets of 10 cores each with HT on (40 threads)*

DEEP LEARNING

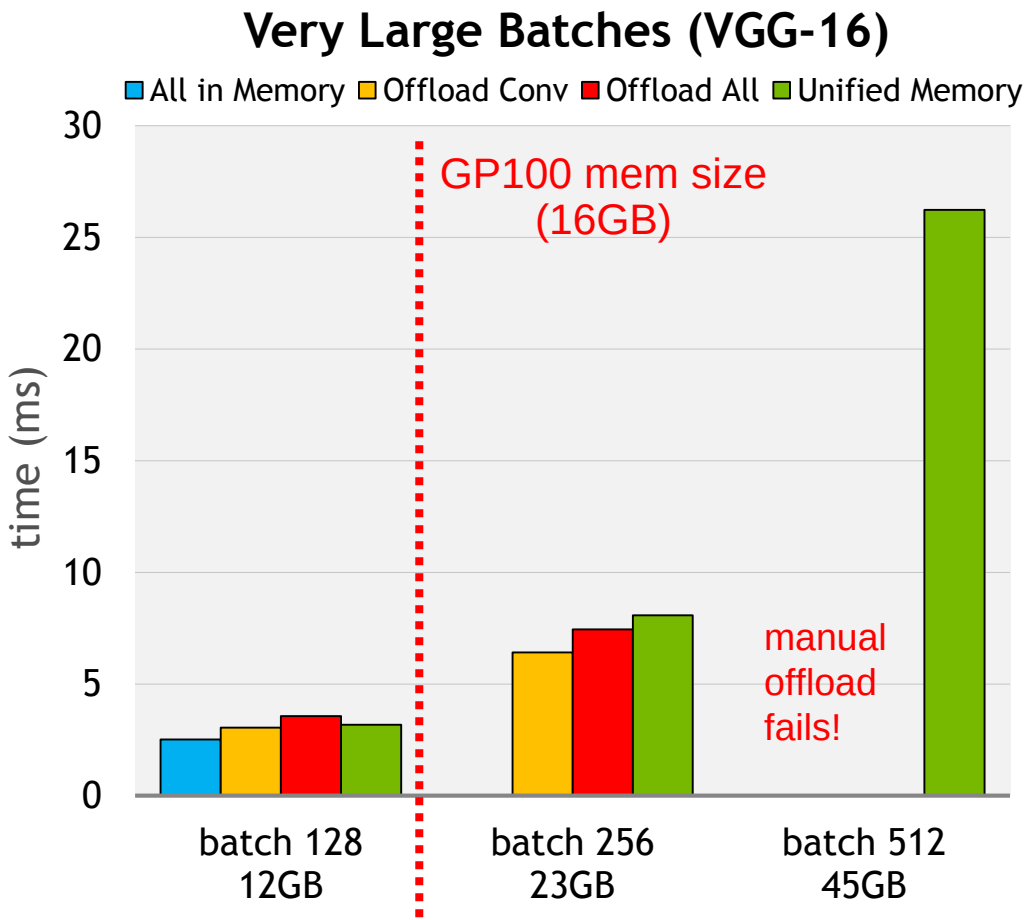
vDNN: Virtualized DNN for Scalable, Memory-Efficient Neural Network Design

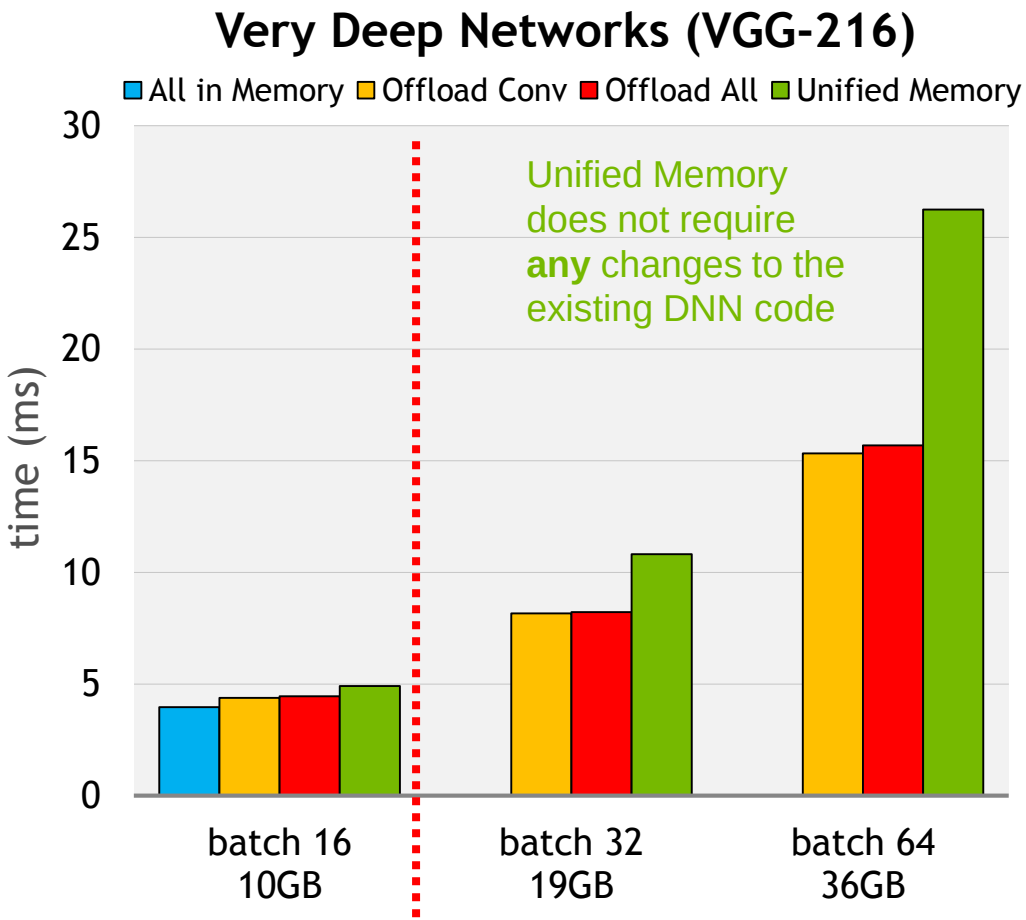


Original version implemented custom heuristics to prefetch and offload data

Unified Memory can automatically migrate memory as needed!

DEEP LEARNING OVERSUBSCRIPTION

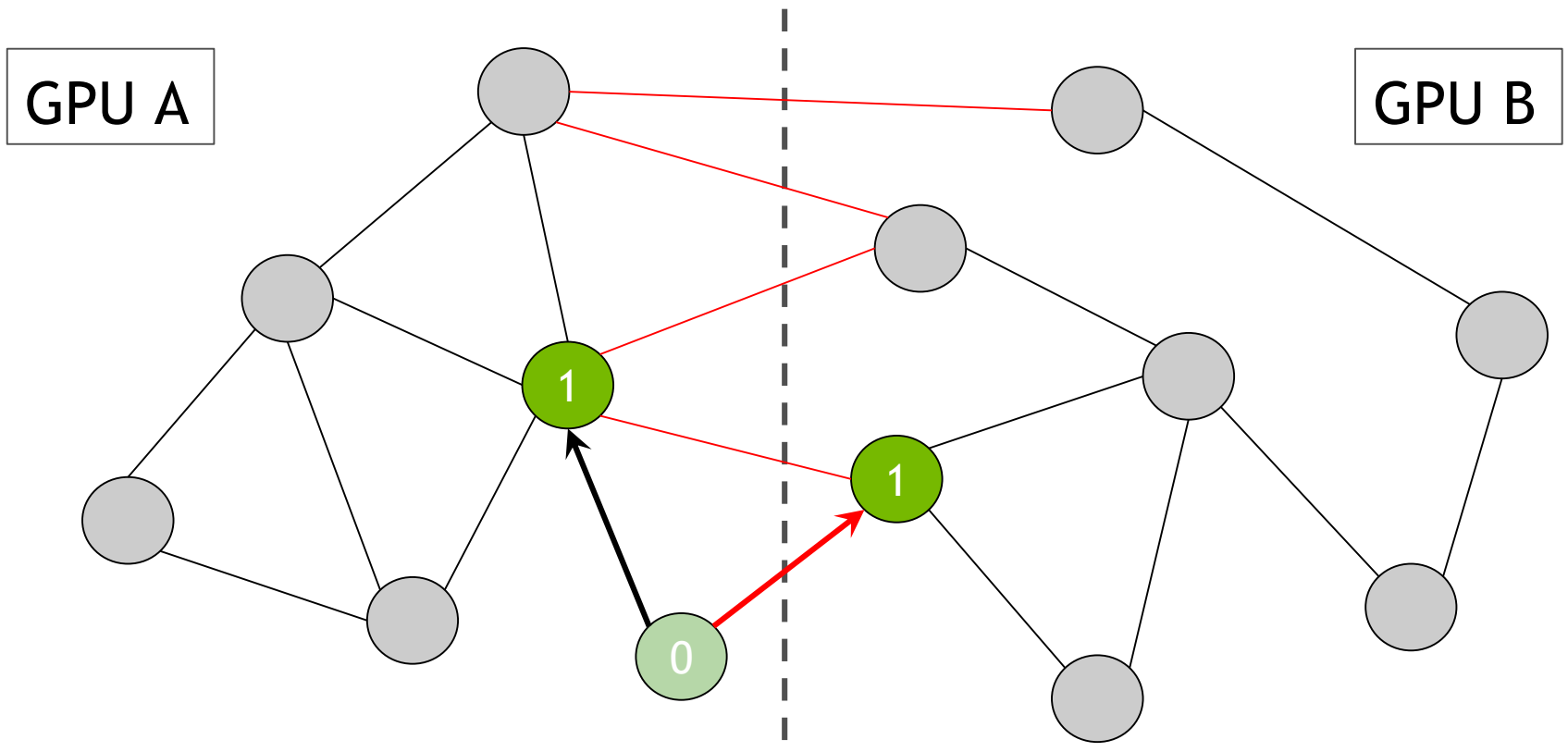




GPU: NVIDIA Quadro GP100; cuDNN 5.1, CUDA 9

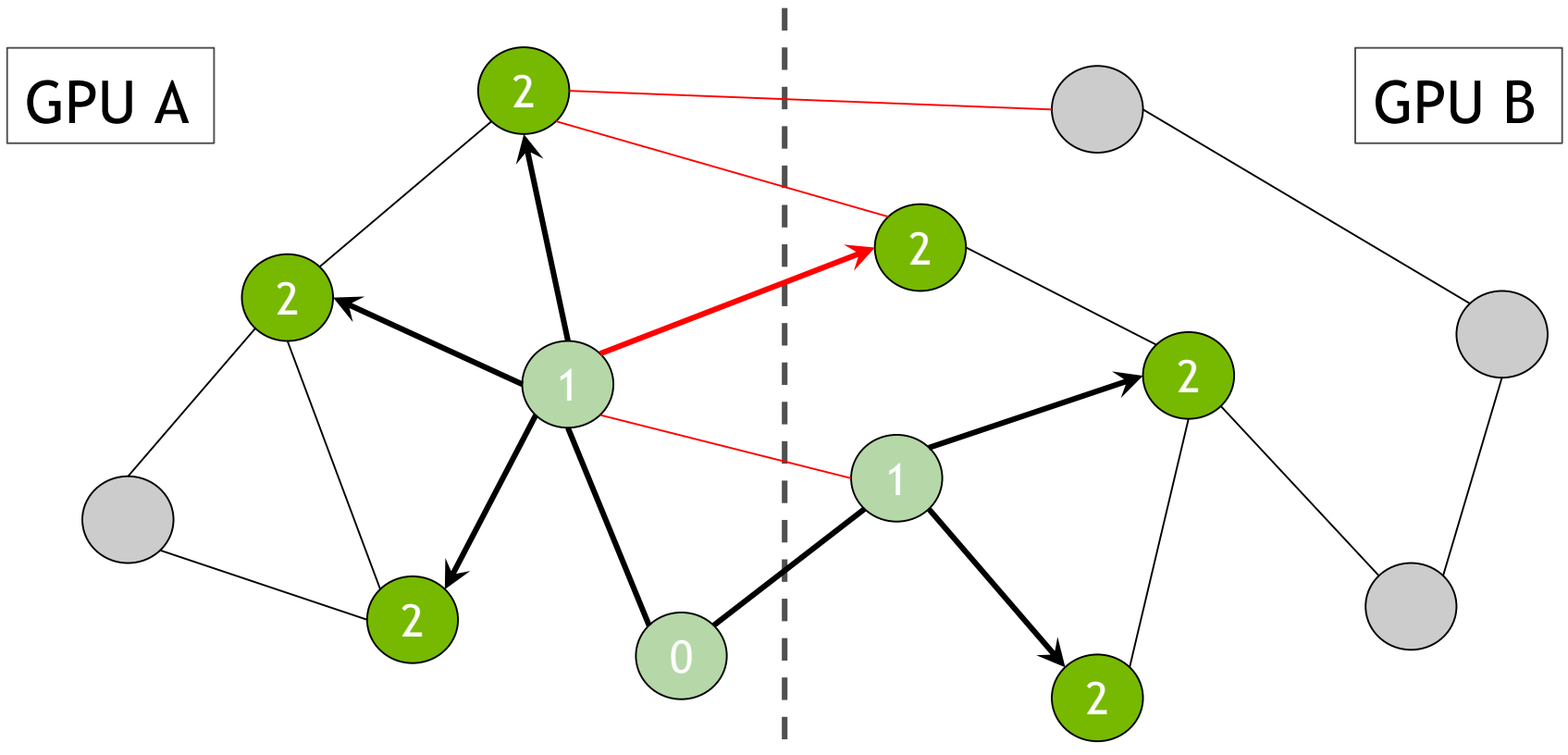
GRAPH ANALYTICS

BFS Traversal



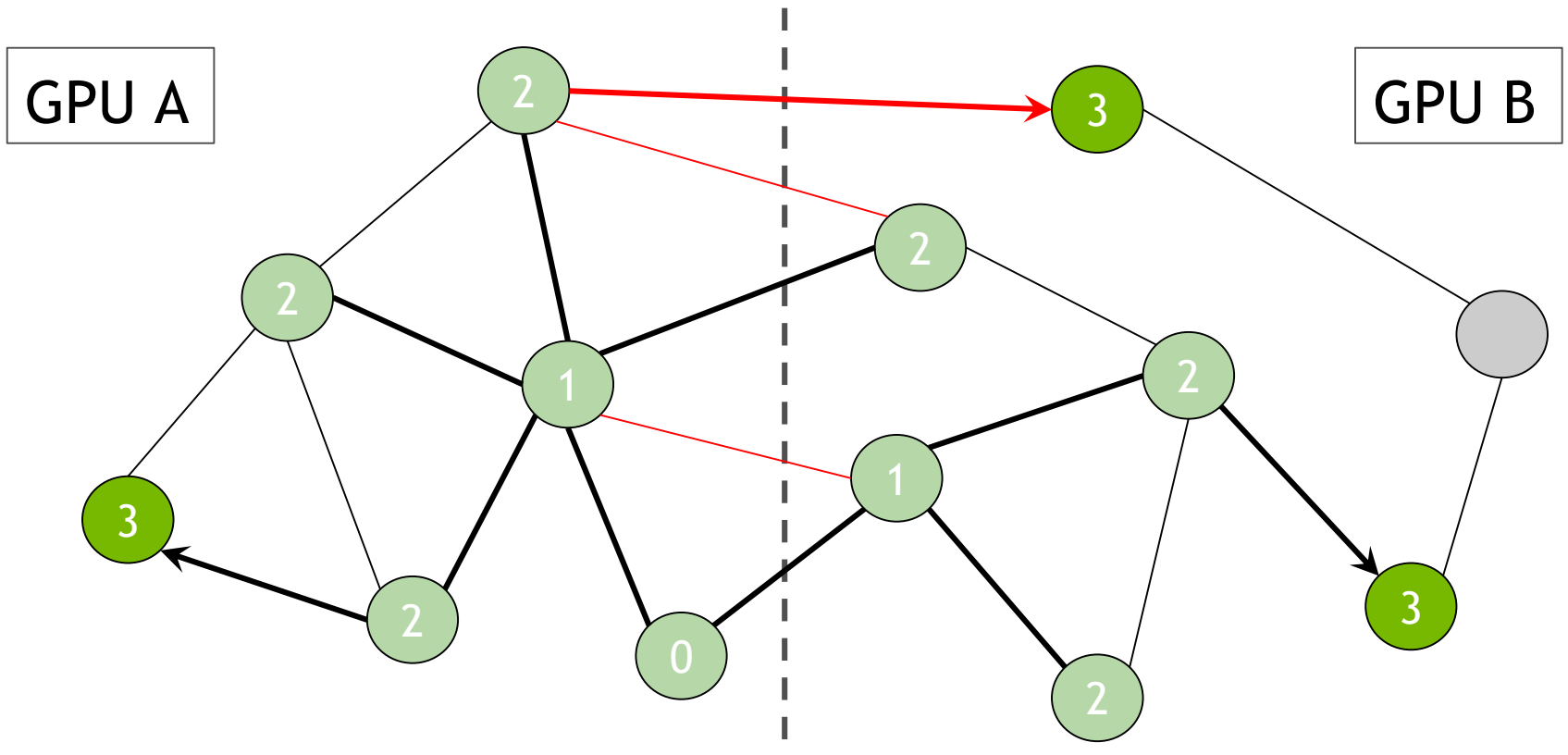
GRAPH ANALYTICS

BFS Traversal



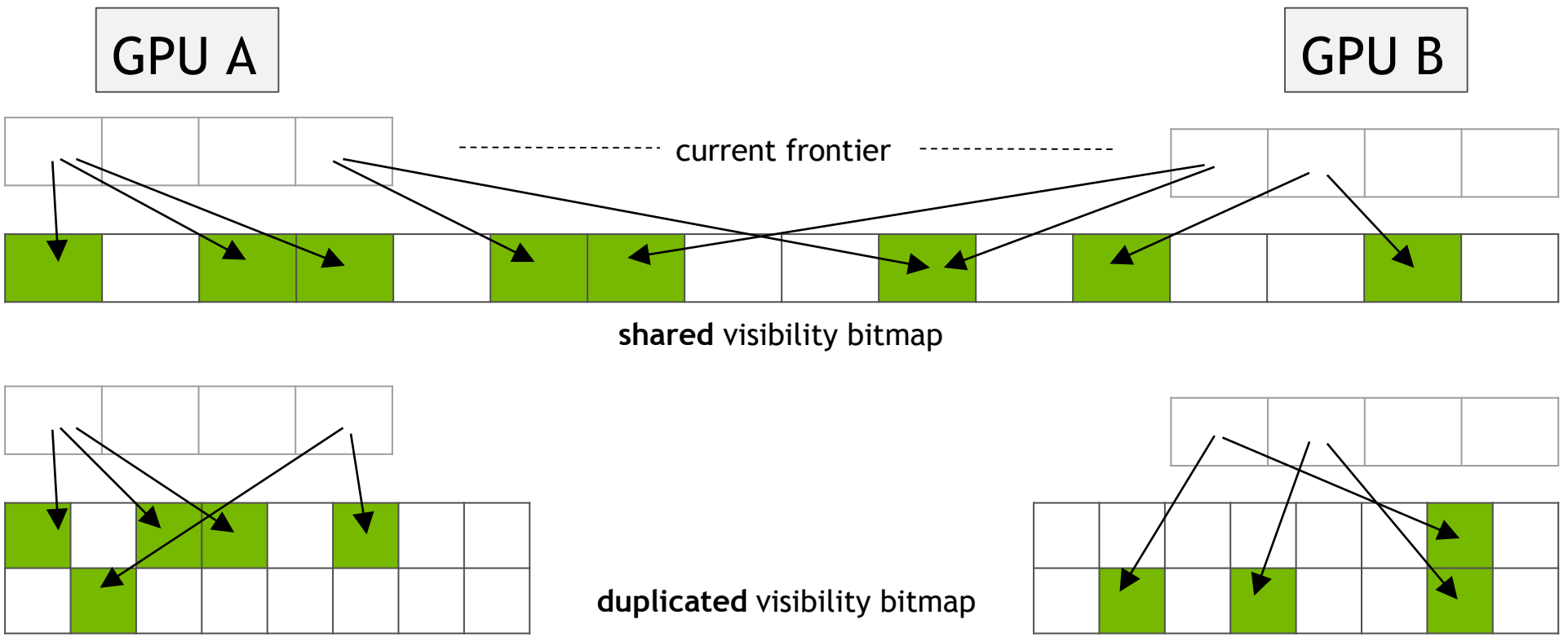
GRAPH ANALYTICS

BFS Traversal



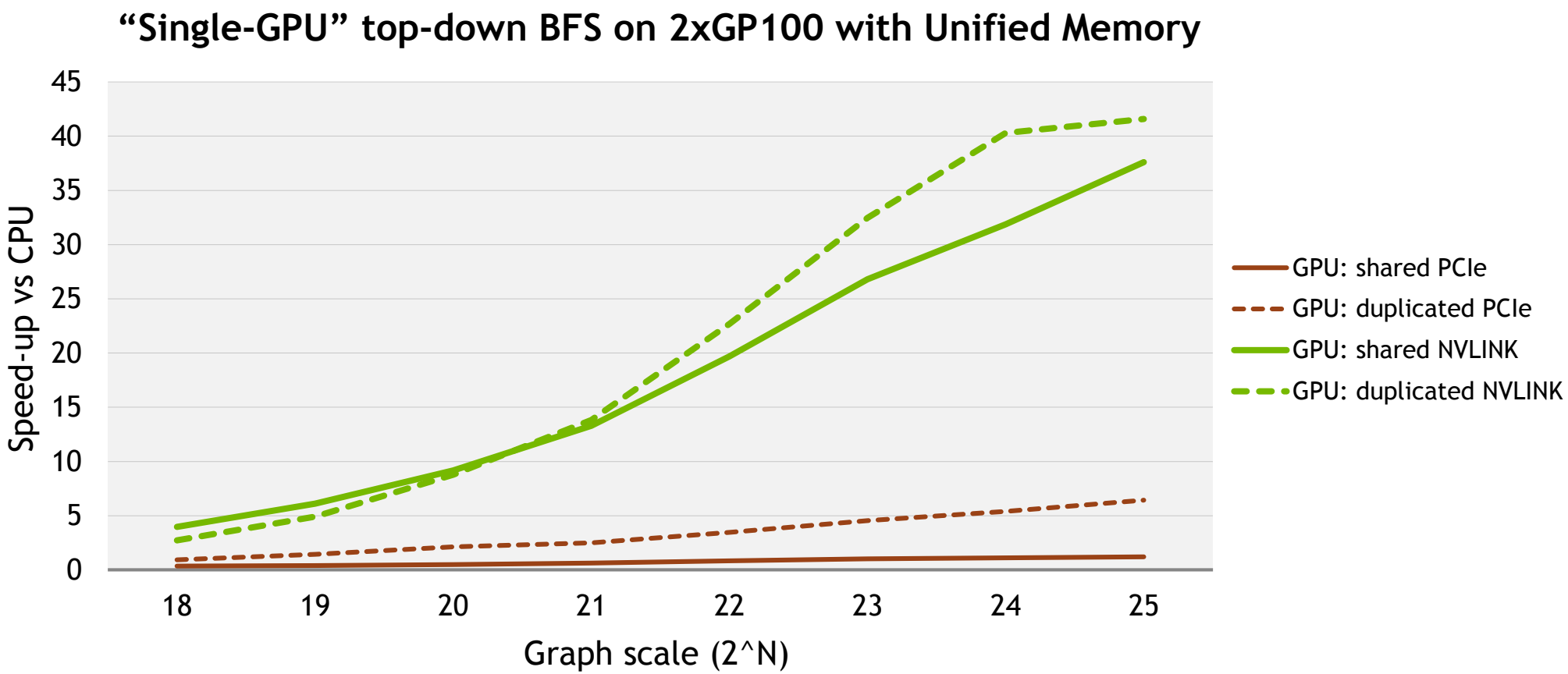
GRAPH ANALYTICS

Shared vs Duplicated Visibility Vector



GRAPH ANALYTICS

Software vs Hardware Atomics



CPU: Intel Core i7-5930K @ 3.50GHz; GPU: NVIDIA Quadro GP100; edgefactor 16, harmonic mean over 64 random sources

AGENDA

Unified Memory Fundamentals

Under the Hood Details

Performance Analysis and Optimizations

Applications Deep Dive

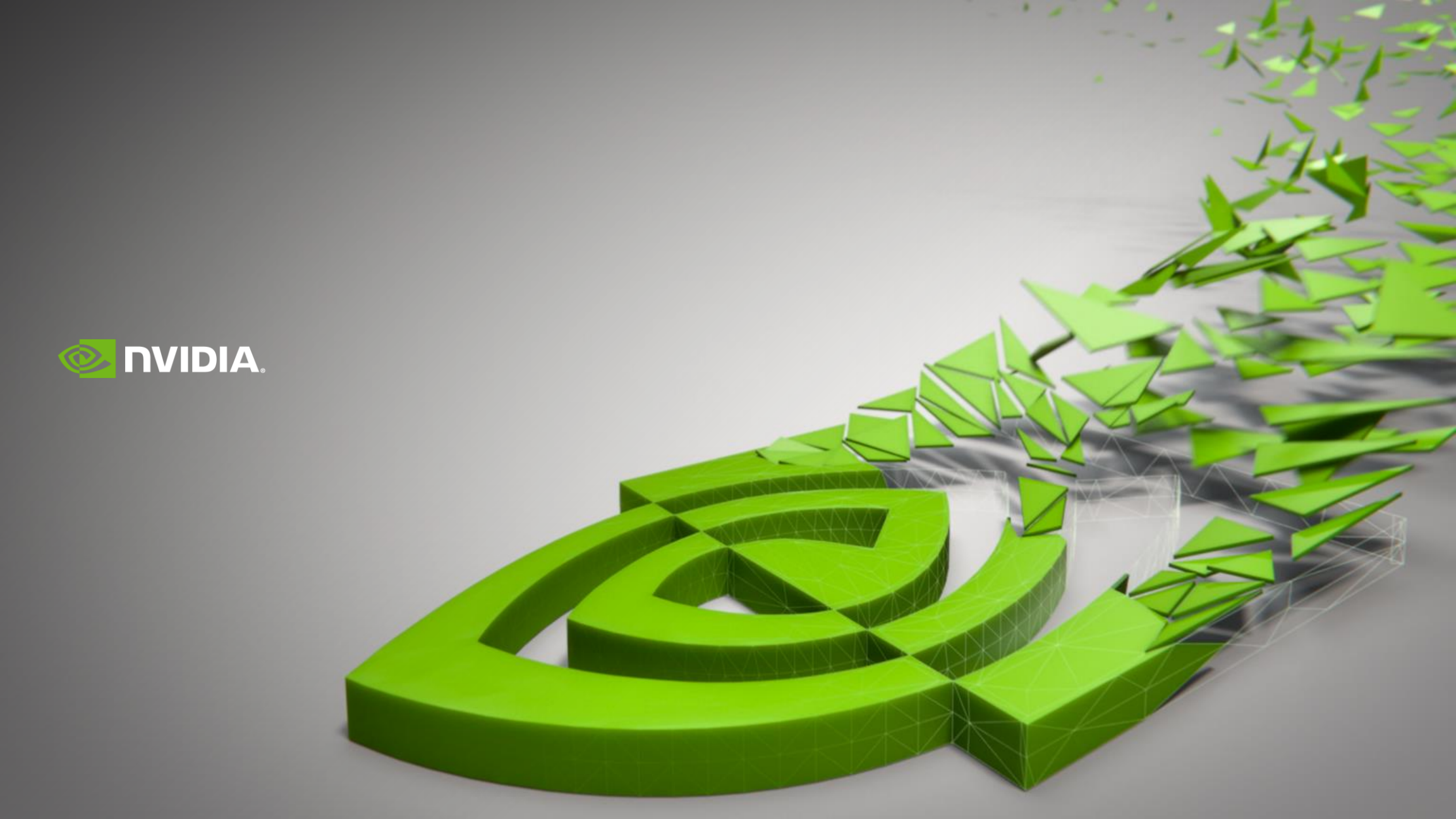
CONCLUSIONS AND OUTLOOK

Consider using Unified Memory for any new application development

- Get your code *running* on the GPU much sooner!
- Enjoy clean code and *virtually* no memory limits

Increase productivity, explore and prototype new algorithms

- Use the explicit data management only *where you need it*


NVIDIA